VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2019 Dominik Berta

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Dominik Berta**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Projektově.CZ s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Lenka Skanderová, Ph.D.**

Konzultant bakalářské práce: Ing. Zdeněk Solnický

Datum zadání: 01.09.2018
Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 15. apríla 2019 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 15. dubna 2019

.....

Na tomto mieste by som rád poďakoval všetkým, ktorí mi s touto prácou pomohli, hlavne Richardovi Římanovi a Ing. Zdeňkovi Solnickému za možnosť získať odbornú prax, množstvo nových poznatkov, ako aj ich ochotu pomôcť pri každom probléme. Zároveň by som chcel poďakovať Ing. Lenke Skanderovej, Ph.D. za vedenie tejto práce.

Abstrakt

Táto bakalárska práca popisuje priebeh mojej odbornej praxe v spoločnosti Projektově.cz na pozícii frontend vývojára v knižnici ReactJS. V úvode je opísaná firma a technológie využité v priebehu praxe. Po nástupe na prax som sa zaoberal vývojom menších vylepšení v systéme. Najväčšia časť tejto práce je venovaná riešeniu hlavnej úlohy - vytvorenie nového Ganttovho diagramu. V závere sa zaoberám hodnotením výsledkov praxe.

Kľúčové slová: JavaScript, React, prax, CSS, HTML

Abstract

This bachelor thesis describes the process of my professional practice in Projektově.cz company on the frontend developer position in ReactJS. The company and the technologies I worked with are described in the introduction. When I started my practice, I was developing smaller system improvements. The biggest part of this work is devoted to solving the main task - creating a new Gantt chart. In the end I deal with the evaluation of the practice results.

Key Words: JavaScript, React, practice, CSS, HTML

Obsah

Zoznam použitých skratiek a symbolov

CSS	–	Cascading Style Sheets
SVG	–	Scalable Vector Graphics
DOM	–	Document Object Model
HTML	–	HyperText Markup Language
JSX	–	JavaScript Extension
URL	–	Uniform Resource Locator

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov zdrojového kódu

1 Úvod

Žijeme v dobe, v ktorej väčšina firiem pri nástupe do zamestnania požaduje okrem kvalitného vzdelania aj nejakú prax, nakoľko štúdium sa často zameriava hlavne na teoretickú prípravu študentov. Možnosť absolvovať bakalársku prácu formou praxe som považoval za ideálnu príležitosť na to, ako získať už popri škole skúsenosť s prácou na reálnom projekte vo firme. To sa mi podarilo vďaka firme Projektově.cz. Nakoľko ide o menšiu firmu, mal som možnosť rozšíriť svoje vedomosti vďaka individuálnemu prístupu skúseného programátora a vyvinúť dôležité časti systému, ktoré sú reálne využívané v praxi, a nie sú iba časťou kódu, ktorý po čase upadne do zabudnutia.

Cieľom tejto práce je opísať priebeh mojej praxe od nástupu až po jej záver. V úvode bude opísaná firma, v ktorej som túto prax vykonával a technológie, s ktorými som počas tohto obdobia pracoval. Nasledujúce kapitoly sa budú venovať menším úlohám, ktorých cieľom bolo naučiť ma základy JavaScriptovej knižnice React a zoznámiť ma s rozsiahlou aplikáciou. Neskôr opisujem zadanie a riešenie mojej hlavnej úlohy zadanej v priebehu tejto praxe – vytvorenie novej interaktívnej komponenty Ganttovho diagramu. V závere práce sú uvedené chýbajúce a získané vedomosti, ako aj celkové zhodnotenie praxe.

2 O firme Projektově.cz s.r.o.

2.1 Základné informácie o firme

Firma Projektově.cz s.r.o. je spoločnosť so sídlom v Ostrave, ktorá vznikla v roku 2013. Hlavnou náplňou pôsobenia tejto spoločnosti je poskytovanie služby Projektově.cz – nástroja pre efektívne riadenie projektov v malých a stredných podnikoch. Spoločnosť spolupracuje s viacerými odborníkmi na projektové riadenie, či už z akademickej pôdy, alebo z partnerských firiem, rovnako tak s univerzitou VŠB – Technická univerzita Ostrava [1].

2.2 Služba Projektově.cz

Táto aplikácia poskytuje nástroje pre efektívne riadenie projektov s vysokým dôrazom na jednoduchosť, prehľadnosť a intuitívnosť.

Služba je poskytovaná podľa modelu SaaS (Software as a Service). Ide o model nasadenia softvéru, pri ktorom užívateľ nemusí inštalovať tento program na svojom počítači, ale k používaniu mu postačuje internetový prehliadač a pripojenie k internetu. Medzi výhody patrí tiež kratší čas potrebný pre nasadenie systému u konkrétneho klienta, nižšie náklady na prevádzku tohto softvéru u klienta, ako aj bezstarostné aktualizácie aplikácie [1].

Backend aplikácie je vyvíjaný v programovacom jazyku Ruby s využitím frameworku Ruby on Rails. Frontendová časť je vytvorená v jazyku JavaScript pomocou knižnice React.

2.3 Pracovné zaradenie

Do firmy Projektově.cz som sa hlásil na pozíciu frontend vývojára v Reacte. Pred začiatkom tejto praxe som absolvoval prijímací pohovor, kde som dostal k vyriešeniu jednoduchú úlohu. Potom som ukázal svoje projekty zo školy a vysvetlil som ich riešenie. Boli mi kladené takisto otázky ohľadom mojich predstáv o praxi a mojich skúsenostiach. Po niekoľkých dňoch ma z firmy kontaktovali, že som splnil ich požiadavky a som prijatý. Následne som dostal za úlohu naštudovať si technológie, s ktorými budem pracovať, nakoľko som sa s väčšinou z nich pred nástupom do spoločnosti nestretol.

Všetky úlohy som vykonával na svojom notebooku, preto bolo v prvý deň mojej praxe nevyhnutné nainštalovať všetko potrebné k vývoju a naučiť sa to používať. Zároveň mi boli odovzdané zdrojové kódy aplikácie. Na začiatku praxe som dostal niekoľko menších úloh, aby som sa naučil pracovať s Reactom a zorientoval sa v systéme, ktorý je pomerne rozsiahly. Hlavne pri prvých úlohách na mňa dohliadal a pomáhal mi skúsený seniorský programátor, s ktorým som neskôr konzultoval aj postup vývoja hlavnej úlohy a prípadné problémy. Ako hlavná úloha mi bolo pridelené vyvinúť novú interaktívnu komponentu Ganttovho diagramu.

3 Použité technológie

V priebehu vykonávania mojej praxe som sa stretol s viacerými technológiami, z ktorých väčšina bola pre mňa nová. Nasledujúce riadky budú venované opisu najdôležitejších z nich.

3.1 React

React je JavaScriptová knižnica vyvinutá spoločnosťou Facebook určená k jednoduchej tvorbe užívateľského rozhrania. Je založená na vytváraní malých zapúzdrených komponent s vlastným stavom, ktoré je možné skladať a vytvárať tak komplexné uživateľské rozhrania webových aplikácií. Efektivita tohto nástroja spočíva hlavne v účinnom obnovení a prekreslení iba tých častí, pri ktorých došlo k zmene dát, bez nutnosti opätovného načítania celej stránky. Na zobrazenie jednotlivých prvkov využíva React plnohodnotný programovací jazyk – JavaScript [2].

3.1.1 JSX

JSX alebo JavaScript extension je rozšírenie Reactu, ktoré nám umožňuje písať HTML kód jednoduchšie a prehľadnejšie priamo do tela JavaScriptových funkcií. Ide o syntaktickú pomôcku pre zápis funkcie *React.createElement*, ktorej použitie je dobrovoľné, avšak doporučené. JSX je možné použiť aj ako návratovú hodnotu, parameter funkcie, prípadne vo vnútri podmienkových blokov, nakoľko sú tieto výrazy kompilované na bežný JavaScriptový kód. Umožňuje tiež definovať atribúty a priradzovať im JavaScriptové výrazy ohraničené zloženými zátvorkami. Použitie tohto rozšírenia pomáha predchádzať injekčným útokom [2]. Na výpisoch 1 a 2 sú porovnané zápisy kódu s využitím JSX a bez jeho využitia.

```
Class Example extends React.Component {
      render() {
            return ( <h2 className="Gantt__heading">Ganttov diagram</h1> );
      }
}
```

Výpis 1: Ukážka zápisu s využitím JSX

```
Class Example extends React.Component {
      render() {
            return React.createElement( 'h2', { className: 'Gantt__heading' },
            'Ganttov diagram' );
      }
}
```

Výpis 2: Ukážka zápisu bez využitia JSX

3.1.2 Komponenta

Komponenty sú základné stavebné bloky každej aplikácie napísanej v Reacte, ktoré umožňujú rozdeliť užívateľské rozhranie do viacerých nezávislých, znovupoužiteľných častí. Za komponentu sa považuje aj JavaScriptová funkcia prijímajúca vstupy a vracajúca element Reactu. Komponenty sa môžu v metóde *return* odkazovať na iné komponenty, a tak vytvoriť svoj výstup zložený z viacerých samostatných častí [2].

Vlastné dáta komponenty sú udržiavané v objekte *state*, ktorého zmeny sa riadia určitými pravidlami. Tento objekt by sme nemali modifikovať priamo (this.state.myState = myNewState), pretože pri takejto zmene nedôjde k požadovanému prekresleniu komponenty. To neplatí iba v prípade nižšie spomenutej metódy *constructor*. Namiesto priamej zmeny objektu obsahujúceho vnútorné dáta komponenty sa využíva metóda *this.setState({myState: 'valueOfMyState'})*. Táto metóda môže meniť *state* asynchrónne, preto by sme sa nemali spoliehať na ich hodnoty pri ďalších výpočtoch vrámci jedného zavolania funkcie *this.setState*. V takomto prípade je lepšie použiť druhú formu tejto metódy, ktorá neprijíma ako parameter objekt, ale funkciu s predchádzajúcim *state*. Nakoľko sa jedná o zapuzdrený datový objekt, nie je možné k nemu pristupovať z komponent iných, ako je vlastník týchto dát. Komponenta môže však tieto dáta poslať do svojich potomkov – ako objekt nazývaný *props*. Takto odoslané dáta by mali slúžiť výlučne na čítanie bez ich následnej modifikácie [2].

3.1.3 Životný cyklus komponenty

Každá komponenta sa vyznačuje tým, že má svoj životný cyklus, vďaka ktorému môžu byť uvoľnené zdroje využívané komponentami, ktoré už nie sú aktívne. Ide o súbor metód ovládajúcich tento cyklus, ktoré sú volané automaticky podľa určeného poradia, a zabezpečujú inicializáciu, vytvorenie, obnovenie a odstránenie príslušnej komponenty [2].

Ako prvé sú volané metódy v čase vytvárania, prípadne po vytvorení inštancie komponenty, v poradí *constructor*, *render* a *componentDidMount*. Metóda *constructor* prijíma *props* a je volaná iba v prípade, ak je nami vytváraná komponenta trieda. V tejto metóde môžeme objektom *state* nastaviť predvolené hodnoty a iba na tomto mieste by sme k nim pri modifikácii mali pristupovať priamo, pomocou *this.state*. Druhým účelom tejto metódy je naviazanie metód na obsluhu udalostí [2].

Ďalšou volanou funkciou je *render*. Je to jediná povinná metóda v triednej komponente. Táto časť komponenty by mala pri každom volaní vrátiť rovnaký výsledok, preto by v nej nemalo dochádzať k žiadnym udalostiam vedúcim k zmene objektu *state*. Úlohou tejto funkcie je vložiť element Reactu do DOM uzla a zobraziť tak výstup danej komponenty na obrazovke. Okamžite po vytvorení komponenty dochádza k volaniu metódy *componentDidMount*, ktorá je vhodná k načítaniu dát. Životný cyklus komponenty obsahuje tiež metódy volané na základe zmien v datách komponenty (*state*), prípadne datách prichádzajúcich (*props*). Najviac využívaná je metóda *shouldComponentUpdate* a metóda *componentDidUpdate* [2].

Štandardne sa každá komponenta prekreslí pri každej zmene dát. Funkcia *shouldComponentUpdate* sa využíva k informácii o tom, či je výstup tejto komponenty ovplyvnený aktuálnou zmenou *state* alebo *props*, čo umožňuje optimalizáciu výkonu aplikácie. V prípade, ak táto funkcia vráti hodnotu *false*, nedôjde k zavolaniu metódy *render*, a teda k prekresleniu komponenty [2].

Pri odstránení komponenty z DOMu je volaná funkcia *componentWillUnmount*, v ktorej sa vykonáva uvoľnenie už nepotrebných zdrojov [2].

3.1.4 Virtuálny DOM

React pri zmene dát neobnovuje priamo reálny DOM, ale k premietnutiu zmien obnoví tzv. virtuálny DOM, čo je kópia reálneho DOMu v pamäti. Pri každej zmene *state* vo vnútri komponenty React nanovo vytvorí celý virtuálny DOM a určitý čas udržiava jeho dve verzie – verziu bez zmeny a verziu so zmenou. Použitím algoritmu *Diff* React porovná obe verzie a určí najmenší počet krokov potrebných k obnoveniu reálneho DOMu. V tomto algoritme využíva predpoklady, čím dosahuje zložitosť O(n). Obnovenie celej virtuálnej štruktúry, následné porovnanie a obnovenie výlučne zmenených častí reálneho DOMu trvá podstatne kratší čas ako obnovenie celého DOM objektu [3].

3.1.5 Kompozícia

Niektoré komponenty napísané v Reacte vytvárajú špeciálne prípady iných komponent, v objektovo orientovaných programovacích jazykoch známe ako dedičnosť. V knihovni React sa táto špecializácia rieši prostredníctvom kompozície. Ide o spôsob, pri ktorom špecifickejšia komponenta vykresľuje komponentu všeobecnú, ktorú nastaví pomocou dát *props*. Majme napríklad všeobecnú komponentu zobrazujúcu na obrazovku dialógové okno. V komponente zobrazujúcej uvítaciu správu po prihlásení môžeme použiť túto všeobecnú komponentu, ktorej zašleme text na zobrazenie. Tá sa potom postará o jeho vykreslenie. Následne pri zobrazení, napríklad správy o odhlásení, môžeme využiť tú istú všeobecnú komponentu, ktorej v objekte *props* odošleme iný text. To nám zaručí napríklad to, že budú obe dialógové okná v rovnakom dizajne a prípadné zmeny v jednom z okien sa premietnú aj v okne druhom. Kompozícia nám teda v Reacte umožňuje využiť znovupoužiteľnosť a vytvoriť zložitejšie celky z jednoduchších častí so zachovaním plnej flexibility, jednoduchosti a jedného z hlavných konceptov tejto knižnice, ktorým je model založený na komponentách [2].

3.2 Flux

Flux je architektonický vzor určený na riadenie toku dát cez aplikácie vytvorené v Reacte, ktorého hlavným princípom je jednosmerný tok dát. Má tri hlavné časti:

- dispatcher
- stores
- views

Tento vzor je založený na interakcii užívateľa s *View* (komponenta vytvorená v Reacte), ktorá je reprezentovaná prostredníctvom akcie. Táto akcia je spracovaná časťou *Dispatcher* a následne šírená ďalej do úložisk (*Stores*), ktoré sú takto informované o vykonaných zmenách. Úložiská uchovávajú aplikačné dáta a logiku, ktoré slúžia na reprezentáciu *Views*, a sú jediným „zdrojom pravdy“. Po tom, ako úložiská vykonajú zmeny vo svojich dátach, vyšlú udalosť o zmene do tzv. *controller-views*, ktoré prijímajú tieto dáta vo vnútri komponent. Prijatie zmenených dát vyvolá prekreslenie samotnej komponenty a všetkých jej potomkov. Takto sa zmena premietne do všetkých *Views*, ktoré sú ňou ovplyvnené [4].

3.3 Immutable.js

Immutable.js je knižnica pre prácu s Reactom navrhnutá k poskytovaniu tzv. Immutable dát. Ide o dáta, ktorých stav nie je možné zmeniť po tom, ako boli už raz vytvorené, čo zjednodušuje vývoj aplikácií. Ak napríklad použijeme metódu *push* na vloženie nového prvku do kolekcie List, tento prvok sa nevloží do už existujúcej kolekcie, ale do jej novo vytvorenej kópie. Pôvodné dáta tak ostanú v nezmenenej podobe. Vďaka využitiu Immutable objektov môžeme predísť programatorským chybám a urýchliť chod aplikácie [5] [6].

3.4 Lodash

Lodash je JavaScriptová knižnica, ktorá umožňuje písať a udržiavať kód jednoduchšie. Metódy Lodashu dokážu pracovať s číslami, objektami, reťazcami, poliami a sú účinným nástrojom pre ich iteráciu, manipuláciu, testovanie a vytváranie zložených funkcií [7].

3.5 Moment.js

Moment.js je knižnica jazyka JavaScript určená na manipuláciu s časom a dátumom. Umožňuje jednoducho analyzovať, overovať a zobrazovať dátumy so zachovaním správneho formátu podľa lokalizácie užívateľa [8].

4 Úlohy zadané v priebehu praxe

Po nástupe do firmy bol prvý deň venovaný inštalácii softvéru potrebného pre vývoj do môjho notebooku, rovnako tiež zoznámeniu so systémom z pohľadu užívateľa. Takisto mi boli vysvetlené základné princípy verzovacieho systému *Git*. Vo firme používajú k správe verzií systém Bitbucket, ktorý umožňuje okrem iného pracovať na vývoji softvéru vo vytvorenej vývojárskej vetve a novovytvorené časti nasadiť do už existujúceho systému až po kontrole a odladení všetkých nedostatkov.

Počas mojej praxe som najčastejšie používal príkazy na vytvorenie novej vetvy, po každej rozsiahlejšej úprave (najčastejšie po každom dni) aj príkazy na potvrdenie a odoslanie vykonaných zmien na server. Keď som svoju prácu ukončil, vytvoril som tzv. *pull request*, kedy sa tento kód odoslal ku kontrole pred nasadením do hlavnej vetvy *master*. Následne mohol môj nadriadený tento kód skontrolovať, okomentovať prípadné nedostatky a vrátiť mi vytvorenú časť k oprave, prípadne k dopracovaniu. Ak sa v kóde nenašli žiadne väčšie nedostatky, ktoré by obmedzovali funkčnosť, nespĺňali požadované štandardy alebo „coding style“ firmy, mohli byť nasadené na produkčné servery spoločnosti. Po zavedení väčších zmien do vetvy master bolo potrebné zaviesť tieto zmeny aj do mojej vývojárskej vetvy príkazom *merge*.

Počas nasledujúcich dní mojej praxe som dostal najprv niekoľko menších úloh, ktorých cieľom bolo v prvom rade zoznámiť sa s knižnicou React a vývojom v nej, osvojiť si základy kaskádových štýlov a zorientovať sa v celom rozsiahlom systéme - odkiaľ a aké dáta prichádzajú, ako ich prípadne spracovať a následne zobraziť užívateľovi v čo najlepšej forme. Každú mnou vytvorenú komponentu som musel upraviť pomocou CSS podľa požiadaviek firmy tak, aby jednotlivé komponenty zapadli do vzhľadu celej aplikácie a spĺňali najnovšie odporúčania ohľadom vnímania rozhrania užívateľom. Tieto úlohy slúžili pre mňa tiež ako úvod do vývoja produktu, ktorý je reálne využívaný stovkami užívateľov denne. Aby som nevytváral pri učení sa iba komponenty, ktoré by sa po dokončení zahodili bez reálneho využitia, dostal som reálne úlohy obsahujúce vylepšenia v systéme. Vytvorené vylepšenia boli neskôr nasadené v hlavnej aplikácii a budú opísané v nasledujúcich riadkoch. V tabuľke 1 uvádzam prehľad zadaných úloh s časom, ktorý som strávil pri ich vypracovaní.

Tabuľka 1: Prehľad strávených časov nad jednotlivými úlohami

Názov úlohy	Strávený čas [Počet dní]
Inštalácia potrebného softvéru	1
Mesačný kalendár	6
Mazanie a radenie blokov na dashboard	4
Správa užívateľov	3
Správa vlastných polí	8
Nový Ganttov diagram	29

4.1 Nová komponenta mesačného kalendára

Prvou mojou úlohou po nástupe na prax bolo vytvoriť komponentu mesačného kalendára na zobrazovanie úloh. Táto časť aplikácie nebola doteraz napísaná v jazyku JavaScript a celé vykreslenie bolo spracovávané na serveri. Cieľom tejto úlohy bolo hlavne zrýchlenie vykreslenia kalendára na obrazovku, vzhľadom k tomu, že bolo spracovanie vykonávané na serveri. V dôsledku neustáleho nárastu užívateľov aplikácie bolo výrazne pomalé oproti spracovaniu v prehliadačoch klientov.

4.1.1 Moja prvá komponenta v Reacte

Na začiatku riešenia tejto úlohy som si vytvoril novú vetvu v Gite a vytvoril som moju prvú triednu komponentu s názvom **Calendar**. Táto komponenta mala za úlohu prípravu dát prichádzajúcich zo servera do štruktúry, ktorá bola najvýhodnejšia k zobrazeniu. Takisto sa stará o vykreslenie hlavičky kalendára, umožnenie posunu dátumov dopredu a dozadu oproti súčasnému mesiacu, ako aj o zabezpečenie jednoduchého filtrovania úloh zobrazovaných v danom mesiaci. Zároveň odosielala dáta „dole“ do dcérskej komponenty **Week**, ktorá bola zodpovedná za vykreslenie jedného týždňa. Inštancie tejto komponenty sa vložili do poľa nachádzajúceho sa v hlavnej komponente a následne sa vo funkcii *render* vykreslili na obrazovku. K tomu, aby som vedel určiť, koľko inštancií typu **Week** je potrebných v zobrazenom mesiaci, bolo potrebné toto číslo vypočítať, čo sa ukázalo ako najväčší problém tejto úlohy.

Väčšina bežných kalendárov vykresľuje v každom mesiaci rovnaký počet týždňov a dni, ktoré do tohto mesiaca nepatria, zobrazí inou farbou. Takto sa niekedy vykreslí celý týždeň naviac bez toho, aby obsahoval nejaký deň zo zobrazeného mesiaca. Požiadavkou v zadaní úlohy ale bolo zobraziť iba tie týždne, ktoré obsahujú nejaký deň patriaci do aktuálne zobrazeného mesiaca. V tejto úlohe som sa zároveň prvýkrát stretol s použitím knižnice *Moment.js*, vďaka ktorej som dokázal zistiť počet týždňov v roku, číslo aktuálneho týždňa a následne s týmito údajmi manipulovať. Pomocou tejto knižnice som vykreslil aj hlavičku, nakoľko podporuje lokalizáciu a dokáže zobraziť dátum vo vhodnom formáte v závislosti od aktuálne zvoleného jazyka. Počet týždňov potrebných na vykreslenie som teda dokázal vypočítať ako rozdiel čísla posledného a čísla prvého týždňa v zobrazovanom mesiaci. Problém ale nastal v mesiaci december v prípade, ak posledný decembrový týždeň nekončil práve v nedeľu, ale v ktorýkoľvek iný deň, celý tento týždeň sa už označoval ako prvý týždeň nového roka. V tomto prípade teda výpočet rozdielu čísel posledného a prvého týždňa dával záporný výsledok, preto som k tomuto výpočtu využil prvý týždeň v mesiaci december, ktorý som odpočítal od počtu týždňov v celom roku. V prípade, ak prvý deň nasledujúceho roka nebol pondelok, pridal som ešte týždeň naviac do decembra, aby sa vykreslil aj spomínaný posledný týždeň starého roka označený už ako číslo jeden.

Po vyriešení tohto problému som do kalendára dodal ovládaciu lištu, pomocou ktorej bolo možné posúvať medzi jednotlivými mesiacmi. K posunu zobrazovaných dátumov som využil opäť funkciu knižnice *Moment.js*, presnejšie funkciu *add*, ktorá prijíma dva argumenty. Ako

prvý argument prijala čiastku, ktorá sa má k aktuálnemu dátumu pridať. Druhým argumentom bola časová jednotka, v mojom prípade hodnota *months* vyjadrujúca mesiace.

Keď už bolo zobrazenie dní funkčné, bolo potrebné pripraviť a následne zobraziť úlohy patriace k zobrazenému mesiacu. K tomu bolo potrebné jednotlivé úlohy zoskupiť podľa dní, k čomu som použil už naprogramovanú funkciu *groupBy*.

Pri vykresľovaní názvu projektu a úlohy do kalendára bolo ešte potrebné nastaviť ich maximálnu zobrazovanú dĺžku, aby neboli tieto názvy príliš dlhé a neprehľadné. To som dosiahol použitím funkcie *truncate* z knižnice *Lodash*. Poslednou úpravou súvisiacou s názvami jednotlivých úloh bolo umožniť užívateľovi kliknutím na úlohu v kalendári zobraziť jej detail v dialógovom okne, čo je ukázané vo výpise 3.

```
<Link to={{pathname: `/issues/${t.id}`, state: { modal: true }}}>
```

Výpis 3: Ukážka otvorenia dialógového okna po kliknutí na príslušný element

Celé zobrazenie kalendára trebalo vytvoriť pomocou responzívneho HTML, použitím gridu s prvkami *Row* a *Col*. Takisto bolo nevyhnutné prostredníctvom CSS upraviť základné zobrazenie – nastaviť farby, umiestnenie, typ písma a iné vlastnosti. Ukážku je možné vidieť na výpise 4.

```
.TasksCalendar {
  &__next {
          float: right;
          margin-right: 0.5em;
          margin-bottom: 1em;
          color: $color-blue;
          cursor: pointer;
  }
}
```

Výpis 4: Ukážka zápisu CSS v komponente Calendar

Na zafarbenie dní som použil podmienené štýly. Ak deň nepatrí do príslušného mesiaca, zobrazí sa šedou farbou. Dnešný deň sa zafarbí na modro. Spôsob použitia je ukázaný na výpise 5.

```
<td className={cn('Week__col', {
   'Week__col--gray': day.month() != month.month(),
   'Week__col--today': day.isSame(moment().startOf('day')) })}
key={i}>
```

Výpis 5: Ukážka zápisu podmienených CSS v komponente Calendar

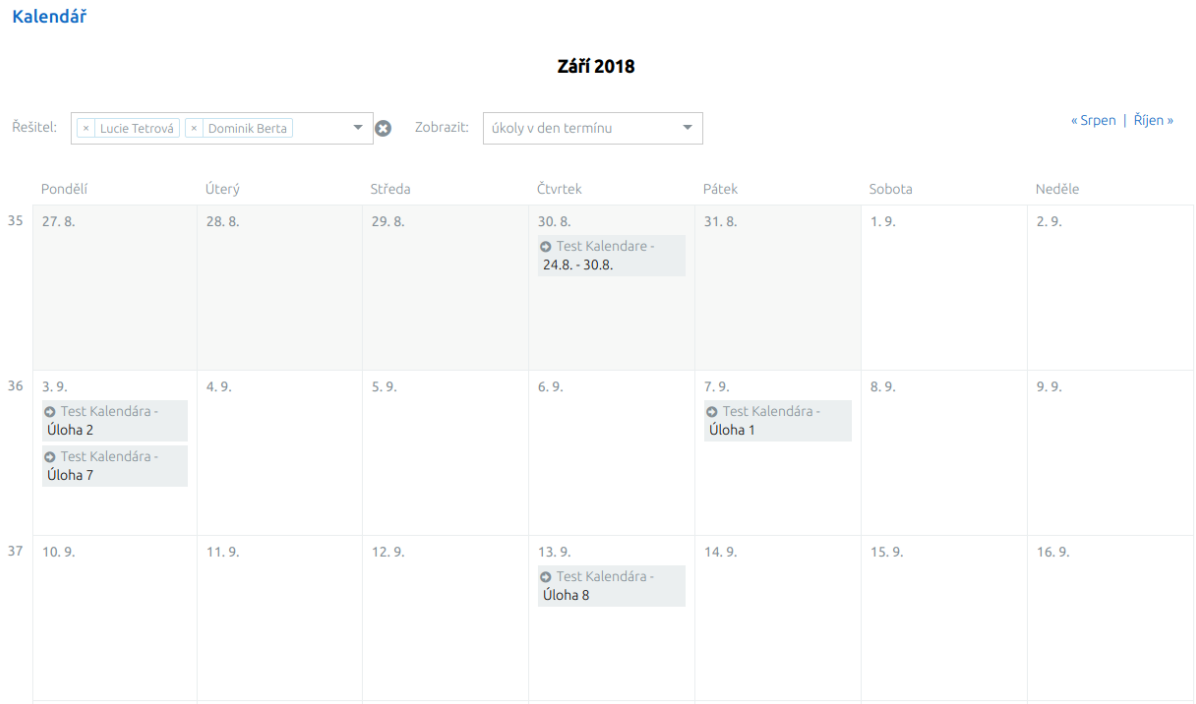
Na záver som do kalendára integroval jednoduché filtre. Jeden filter slúžil k zobrazeniu úloh iba od vybraných užívateľov. Druhý filter pridával možnosť zobraziť úlohy iba v deň termínu, resp. v každom dni ich trvania. Táto funkcia zahŕňala aj ukladanie zvolených filtrov do URL s ich následným načítaním, k čomu som použil už vytvorené funkcie, do ktorých ale bolo potrebné zaslať upravené dáta tak, aby spĺňali požadovanú štruktúru. K vykresleniu samotných filtrov som použil hotovú komponentu *SelectField*, ktorá prijímala jednotlivé položky výberu vo formáte objektu obsahujúceho *label* a *value*. Spomenutú štruktúru som si vopred pripravil prejdením všetkých užívateľov prichádzajúcich z úložiska pomocou funkcie *map* a následného zoradenia podľa abecedy, čo je možné vidieť na výpise 6.

```
let menuItems = this.props.users.map(u => ({ label: u.fmtLong, value: u.id })).
    sortBy(u => u.label, (a, b) => strComparator(a, b)).toArray()
```

Výpis 6: Ukážka prípravy dát pre uloženie filtra do URL

Keďže sa jedná o aplikáciu s možnosťou výberu zo štyroch jazykových verzií, potreboval som preložiť všetky texty nachádzajúce sa v kalendári, vložiť ich do slovníka aplikácie a zobraziť slovníkovým prekladom pomocou knižnice *I18n*.

Vytvorenie podobnej komponenty by možno nebolo náročné pre skúseného vývojára v Reacte, mne však v začiatkoch dala celkom zabrať. Naučil som sa na nej ale všetky základy potrebné pre prácu s touto knižnicou, základy CSS, ako aj základy práce s Gitom.



Obr. 1: Výsledný kalendár vytvorený v Reacte

4.2 Mazanie a radenie blokov na dashboard aplikácie

Mojou druhou úlohou bolo pridať tlačidlo umožňujúce odstrániť bloky na dashboarde aplikácie. Ide o akúsi nástenku, ktorá sa zobrazí po prihlásení a jej úlohou je zobrazovať najnovšie aktualizácie a kľúčové dianie. V službe Projektově je možné si túto nástenku prispôsobiť podľa vlastných potrieb a pridať si tam nastaviteľné bloky v závislosti od požiadaviek každého užívateľa. Na tomto mieste je možné umiestniť napríklad strávené časy, úlohy s konkrétnym riešiteľom, poslednú aktivitu, obľúbené projekty a mnohé ďalšie. Tieto bloky je možné pridávať po jednom, chýbala však možnosť odstrániť nejaký blok v prípade, ak ho chcel užívateľ napríklad nahradiť blokom iným. To bolo možné vykonať iba resetovaním všetkých blokov a následným ručným pridávaním každého bloku samostatne odznova, čo mohlo trvať značný čas. Takisto chýbala možnosť týmto už pridaným blokom zmeniť poradie a určiť, čo sa zobrazí na vrchu tejto nástenky a čo naopak úplne dole. Usporiadanie blokov bolo určené postupnosťou ich pridávania. Mojou úlohou bolo teda pridať tlačidlo s ikonou krížika, ktoré po stlačení odstráni príslušný blok a umožniť radenie týchto blokov prostredníctvom techniky „ťahaj a pusť“ (angl. *drag and drop*).

4.2.1 Prvá skúsenosť s vytváraním drag and drop

Na začiatku riešenia som musel dôkladne naštudovať už vytvorený kód vykresľovania blokov, aby som vedel, kam pridať ikonu krížika tak, aby som nenarušil fungovanie celej časti aplikácie. Keďže sa každý blok vykresľoval v cykle *„map“* vnorením už vytvorenej komponenty **Block**, nebolo potrebné zisťovať detaily vykreslenia každého typu bloku, ktoré boli riešené vo vnútri tejto komponenty. Mazanie blokov malo byť umožnené iba v režime úprav, ktorý som nakoniec tiež trocha upravil. Pridal som teda nový *state* reprezentujúci tento režim úprav, ktorým som podmienil vykreslenie ikony. Po kliknutí na ikonu je volaná funkcia *removeBlock*, ktorá odstráni príslušný blok z databázy prostredníctvom zaslania názvu bloku na server metódou *HTTP POST*. Ikonu bolo potrebné upraviť kaskádovými štýlmi tak, aby sa po vstupe kurzora do jej oblasti zmenila farba krížika na červenú, čo malo indikovať užívateľovi, že sa kurzor nachádza práve nad týmto ovládacím prvkom. To som dosiahol použitím selektora *hover* v CSS. Najťažšou úlohou bolo túto ikonu správne umiestniť pomocou kaskádových štýlov tak, aby sa nachádzala presne tam, kde mala.

Pre druhú časť tejto úlohy bolo potrebné si naštudovať spôsob vytvorenia funkcionality *„drag and drop“*, ktorú som v tejto úlohe riešil pomocou funkcií HTML5. Na rovnakom mieste, kde som vkladal krížik na odstránenie bloku, som najprv pridal *div* element s atribútom *draggable*, ktorý bol podmienený stavom signalizujúcim režim úprav. Na tento element som takisto naviazal udalosť *onDragStart*, kde som definoval činnosti, ktoré sa majú vykonať pri začatí ťahania bloku za účelom jeho premiestnenia. V metóde *onDragStart* som si uložil do *state* index posúvaného bloku, ktorý bolo potrebné uložiť aj do objektu *dataTransfer* pomocou metódy *setData*. Jedná sa o objekt uchovávajúci dáta „drag eventu“ dostupné do ukončenia ťahania, teda do vyvolania

udalosti *onDrop*. Uložený index som neskôr využil v metóde vyvolanej práve vyššie spomenutou udalosťou *onDrop*.

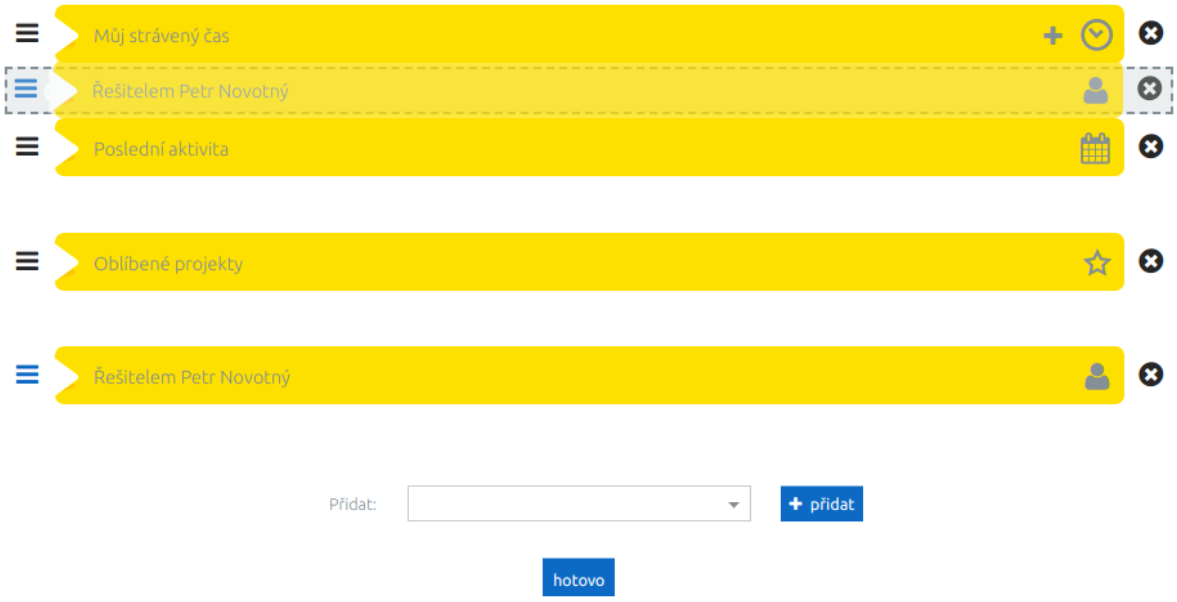
Po vykonaných úpravách som na začiatok a koniec bloku umiestnil tzv. *„drop target“*, teda element, na ktorom je možné ukončiť udalosť *„drag and drop“*. Tieto elementy bolo potrebné ešte vizualizovať po vstupe kurzora s ťahaným blokom do ich vnútra, aby užívateľ vedel, kam môže premiestňovaný blok umiestniť. K tomuto účelu som využil udalosti *onDragEnter* a *onDragLeave*, ktoré sú vyvolané pri vstupe resp. výstupe z elementu, kam ich vložíme. V metóde zavolanej pri vstupe do príslušného priestoru som musel ešte vyriešiť to, aby sa „drop“ zóna nezobrazila aj v priestore aktuálne presúvaného bloku. To som dosiahol porovnaním indexu ťahaného bloku s indexom zóny, do ktorej priestoru som vstúpil. Nakoľko tieto zóny určené k ukončeniu presunutia boli dve – nad a pod každým blokom, bolo potrebné porovnať aj index bloku zvýšený o jeden. Ak sa táto zóna nenachádzala v blízkosti ťahaného bloku, jej index som uložil do *state* s názvom *visibleDropZone*. Vďaka tejto informácii som potom pri vykresľovaní mohol použiť podmienené kaskádové štýly k ovplyvneniu zobrazenia správnych zón.

V metóde *onDragLeave* som zistil, či sa pri opustení jedná o práve zobrazovanú „drop“ zónu a ak to tak bolo, *state* obsahujúci index tejto zóny som nastavil na hodnotu *null*. Do toho istého elementu som umiestnil tiež udalosť *onDrop* vyvolanú pri pustení tlačidla myši počas premiestňovania bloku, v ktorej som z vyššie spomenutého objektu *dataTransfer* „vytiahol“ index presúvaného bloku pomocou funkcie *getData*. Druhý potrebný údaj – index „drop“ zóny bol argumentom funkcie *onDrop*, vďaka čomu som získal oba indexy potrebné k premiestneniu bloku. Do pomocnej premennej som si uložil aktuálne poradie blokov prostredníctvom tzv. *spread* operátora. Presúvaný blok som z tohto poľa odstránil metódou *delete*. Vzhľadom k tomu, že sa tento prvok mohol nachádzať na akomkoľvek indexe, v poli vzniklo „prázdne miesto“. To som odstránil použitím Lodashovej funkcie *compact*, ktorá posunie prvky poľa, na ktoré je aplikovaná. Keďže som vedel index „drop“ zóny, kam mám blok presunúť, pomocou funkcie *splice* som na toto miesto v poli vložil presúvaný blok tak, ako je možné vidieť vo výpise 7.

```
if (index1 > 0) {
   right.splice(index2, 0, elementForInsert)
} else {
   right.splice(index2-1, 0, elementForInsert)
}
```

Výpis 7: Ukážka vloženia prvku do poľa na konkrétny index

Na záver som všetky tieto bloky v správnom usporiadaní odoslal na server opäť prostredníctvom funkcie *HTTP POST*. Ukážka presúvania blokov je na obrázku 2.



Obr. 2: Ukážka premiestnenia bloku prostredníctvom drag and drop funkcie

Táto úloha bola pre mňa náročnejšia jednak tým, že išlo o vytvorenie funkcionality „drag and drop“, s ktorou som sa dovtedy nikdy nestretol, ako aj tým, že bolo potrebné tieto zmeny zakomponovať do už existujúceho kódu. Obidve tieto veci bolo potrebné si dôkladne naštudovať, no vďaka tomu ma opísaná úloha veľa naučila. Toto vylepšenie je v súčasnosti nasadené na produkčných serveroch a verím, že užívateľom ušetrilo nejaký čas a spríjemnilo ovládanie aplikácie.

4.3 Správa užívateľov

Po dokončení mojej druhej úlohy som dostal za úlohu malé vylepšenia v správe užívateľov. Úloha bola rozdelená na tri menšie podúlohy – zobraziť externých sledujúcich, zobraziť „spinner“ pri vytváraní užívateľa a pridať možnosť administrátorovi resetovať heslo ktorémukoľvek z užívateľov, ktorých spravuje.

V systéme Projektově vznikla na podnet užívateľov možnosť zapojiť do úloh vo firmách aj zákazníkov, ktorí pochopiteľne nemajú prístup do systému. Títo užívatelia dostali pomenovanie externí sledujúci a k ich pripojeniu k úlohe stačí vyplniť formulár s e-mailom. V úlohách, kam ich firma pozve, majú následne obmedzené funkcie, jedná sa spravidla o možnosti vidieť názov úlohy, popis úlohy, ako aj svoje komentáre. Čo ale mnohým firmám pri tejto funkcii chýbalo, bola možnosť zobraziť neregistrovaných užívateľov všetkých na jednom mieste, ku každému mať možnosť zobraziť úlohy, v ktorých je zapojený s možnosťou jeho odstránenia z konkrétnych úloh, prípadne z celého systému.

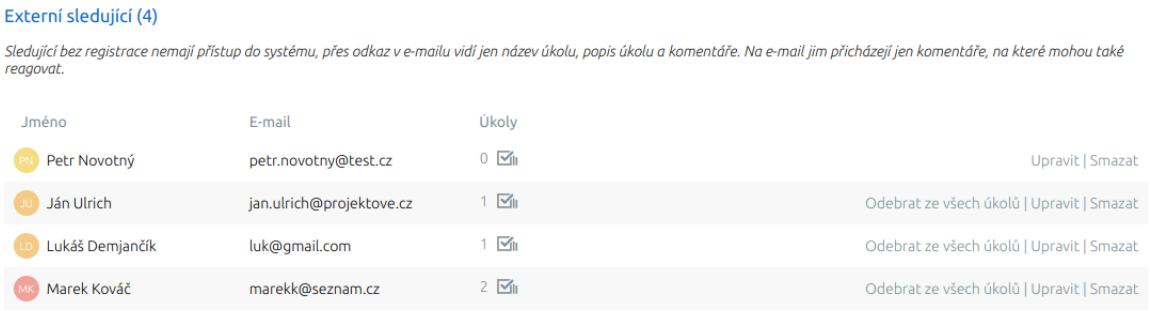
Druhou úlohou bolo zobraziť „spinner“ indikujúci ukladanie užívateľa na server, nakoľko toto odosielanie dát môže chvíľu trvať. Doteraz užívateľ potvrdil pridanie nového užívateľa a nebol nijakým spôsobom informovaný o tom, že sa na pozadí niečo deje a je potrebné chvíľu čakať.

Tretia úloha vznikla rovnako na podnet užívateľov – možnosť vykonať reset hesla administrátorom ktorémukoľvek zamestnancovi.

4.3.1 Riešenie úlohy

Pri riešení tejto úlohy som začal s výpisom externých sledujúcich. Tých som získal zo servera funkciou *HTTP GET* a uložil do *state* s názvom *external*. Získaných užívateľov som následne prechádzal pomocou funkcie *map*, v ktorej som pre každého užívateľa vytvoril komponentu **UserRow**, kam som odoslal príslušné údaje prostredníctvom *props*. Táto komponenta sa starala o výpis externých užívateľov na obrazovku. V momente riešenia tejto úlohy som pre výpis použil už existujúcu komponentu, ktorá sa dovtedy starala iba o výpis registrovaných užívateľov, čo sa neskôr ukázalo ako nie najvhodnejšie riešenie z hľadiska prehľadnosti kódu. So skúsenosťami, ktoré som neskôr počas praxe nadobudol, by som už takmer s istotou volil vytvorenie novej komponenty pre týchto externých sledujúcich.

Vzhľadom k tomu, že sa stĺpce s informáciami zobrazené pri registrovaných a externých užívateľoch líšili, v metóde *return* som tieto odlišnosti musel vyriešiť pomocou ternárnych operátorov využívajúcich informáciu o tom, o aký druh užívateľa sa jedná. Pri externých sledujúcich som zobrazil meno užívateľa, email, počet úloh, v ktorých je zapojený, ako aj možnosti upraviť a zmazať užívateľa. Takisto sa tu zobrazuje možnosť odobrať užívateľa zo všetkých úloh, ktorá sa ale vykreslí iba v prípade, ak tento užívateľ je do nejakých úloh zapojený. Výpis externe sledujúcich užívateľov je možné vidieť na obrázku 3.



Obr. 3: Zobrazenie externe sledujúcich užívateľov

Po kliknutí na ikonu indikujúcu úlohy som vytvoril dialógové okno zobrazujúce jednotlivé úlohy, ktorých je konkrétny užívateľ súčasťou, s možnosťou odstránenia sledovania samostatne pre každú úlohu. Ukážka tohto okna je na obrázku 4. Pri odstraňovaní sledujúceho zo všetkých úloh sa zobrazí takisto dialógové okno, tentokrát s upozornením, a túto akciu je pred vykonaním

nutné potvrdiť. V prípade potvrdenia sa postupne v cykle zasielajú na server požiadavky na odobratie užívateľa z každej úlohy, pri ktorej je evidovaný.

Obr. 4: Zobrazenie úloh externe sledujúceho užívateľa

Druhú úlohu som vyriešil jednoducho pridaním state *spinner*, ktorý som na začiatku metódy zavolanej po stlačení tlačidla „uložiť“ nastavil na hodnotu *true*. Pretože v naviazanej metóde prebiehala aj validácia vstupov, v každej podmienke, ktorej výsledkom bolo vypísanie neúspešného uloženia na obrazovku, som tento *state* nastavil na hodnotu *false*. V prípade, že vykonávanie metódy dospelo až k záverečnej podmienke, zavolala sa funkcia, ktorá dáta o novom užívateľovi odoslala k uloženiu na server. Akonáhle bolo uloženie na serveri úspešné, v metóde *then* som opäť nastavil tento *state* na *false*. To isté som pridal aj do metódy *catch*, aby sa indikátor ukladania nezobrazoval vo chvíli, kedy ukladanie zlyhá. Na záver bolo ešte potrebné zamedziť možnosti opätovného stlačenia tlačidla „uložiť“ v čase, kedy dochádza k ukladaniu, aby to užívateľa zbytočne nemiatlo. Preto som do elementu *button* pridal atribút *disabled*, ktorý bol ovplyvňovaný práve prostredníctvom state *spinner*.

V tretej úlohe som zaviedol *state* reprezentujúci zobrazenie dialógového okna na resetovanie hesla užívateľa, ktorý sa nastavil na hodnotu *true* po kliknutí na príslušné tlačidlo. V tej chvíli sa administrátorovi zobrazí na obrazovke dialógové okno s potvrdením požadovanej akcie a informáciou o odoslaní odkazu na e-mail kolegu. Po potvrdení sa tento požiadavok odošle na server, ktorý sa postará o zaslanie e-mailu na príslušnú adresu.

Vo všetkých prípadoch bolo potrebné jednotlivé elementy upraviť pomocou kaskádových štýlov tak, aby doplnili vzhľad stránky určenej na správu užívateľov. Všetky pridané texty som na záver pridal do slovníka a výsledný kód som odoslal ku kontrole.

4.4 Správa vlastných polí

V tejto úlohe som dostal zadanie vytvoriť prehľadný nástroj na spravovanie tzv. vlastných polí. Ide o polia, ktoré je možné pridávať k úlohám a projektom a ich cieľom je štruktúrovane uchovať základné informácie o projekte. Tieto polia sa v každej firme líšia a môžu obsahovať napríklad meno zodpovednej osoby, odkaz na hlavný súbor k projektu alebo čokoľvek potrebné pre konkrétnu spoločnosť. Pretože doteraz priamo v systéme dostupnom užívateľom neexistoval nástroj, ktorým by si mohli tieto polia pridávať, upravovať a nastavovať podľa potrieb, boli užívatelia nútení písať na podporu aplikácie, kde im človek zabezpečujúci obsluhu podpory nastavil požadované polia. Z tohto dôvodu bolo potrebné vytvoriť intuitívne rozhranie, pomocou ktorého by zvládli nastaviť tieto polia správcovia systému vo všetkých firmách.

4.4.1 Riešenie

Prvým krokom bolo naštudovanie súčasného nástroja na nastavovanie vlastných polí. V novovytvorenej komponente som do gridu umiestnil názov a pripravil si tabuľku, do ktorej som neskôr vykresľoval jednotlivé polia – každý riadok pre iné pole. Keďže sa vlastné polia týkajú samostatne úloh a projektov, bolo nevyhnutné pridať nejaký spôsob prepínania medzi zobrazovanými nastaveniami. To som dosiahol použitím prvku *Tab*, ktorého atribút *name* sa zhodoval s obsahom state *active* (hodnota *tasks* alebo *projects*), vďaka čomu bolo možné rozhodnúť, ktorá záložka má byť aktívna.

Potom som začal spracovávať dáta prichádzajúce zo *store*, aby som mohol vypísať už vytvorené polia a k nim potrebné informácie. Dáta prichádzali do komponenty v zložitej štruktúre, ktorú som musel niekoľkými krokmi upraviť do požadovanej podoby. V prípade úloh bolo potrebné zobraziť stĺpce v nasledujúcom poradí: názov poľa, fronta, typ poľa, radenie a tlačidla na úpravu a odstránenie príslušného poľa. V prípade projektov to bolo podobné, akurát stĺpec fronta bol nahradený atribútom viditeľnosť polí. Tento atribút uchováva typy rolí, ktoré môžu príslušné pole vidieť pri konkrétnom projekte.

K výpisu konkrétneho riadku obsahujúceho jedno pole som si vytvoril novú komponentu s názvom **FieldRow**, v ktorej som pomocou tabuľky zobrazil všetky potrebné informácie. Do predposledného stĺpca som umiestnil ikony šípiek na posúvanie polí vyššie a nižšie v zobrazovanej tabuľke. Po ich kliknutí sa tento posun odoslal na server.

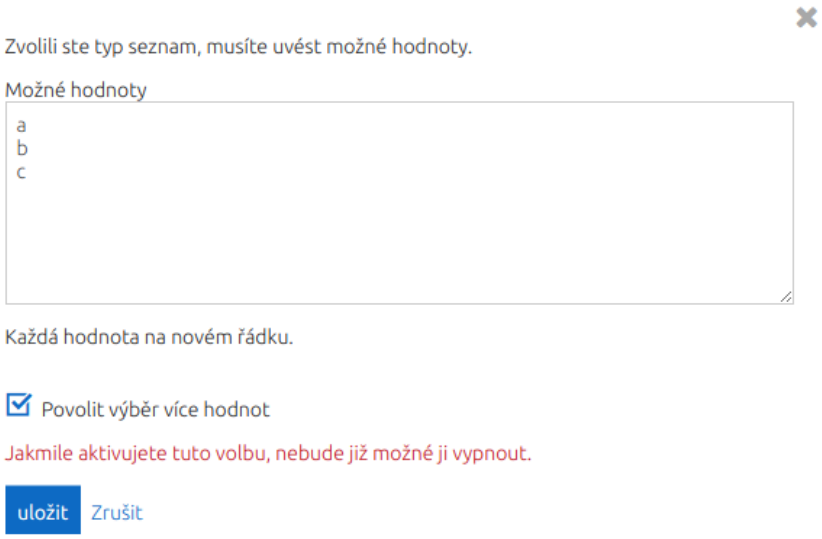
Po zobrazení dát prichádzajúcich zo servera som začal pracovať na možnosti pridania nového poľa po stlačení tlačidla „pridať ďalšie pole“. To som trocha nešťastne riešil v tej istej komponente ako zobrazovanie už vytvoreného poľa, čo viedlo k príliš dlhému kódu v metóde *return*. Som presvedčený, že po ďalších skúsenostiach získaných v priebehu praxe by som toto vykreslenie riešil spôsobom samostatnej komponenty. V metóde *return* som teda zistil, či sa v prijatých *props* dátach nachádza názov poľa a podľa atribútu *name* som s pomocou ternárneho operátora určil, čo sa vráti do rodičovskej komponenty a vypíše na obrazovku. Nakoľko všetko okrem názvu

a popisu poľa je možné vybrať iba z určitých možností, k tomuto nastaveniu som využil prvok *SelectField*, ktorého obsah som si vopred nadefinoval.

Pred uložením nového poľa na server bola potrebná ešte validácia zadaných hodnôt, vzhľadom k tomu, že názov a typ poľa sú povinné položky, bez ktorých nie je možné nové pole vytvoriť. Pred zaslaním na server som teda pomocou podmienok odkontroloval každý povinný stĺpec ako aj každú ich kombináciu samostatne, kvôli schopnosti zobraziť užívateľovi čo najrelevantnejšiu hlášku o tom, prečo uloženie zlyhalo a čo je potrebné upraviť. Pri položkách pridaného poľa som v prípade vypĺňania názvu musel skontrolovať tiež situácie, kedy by užívateľ zadal iba prázdnu medzeru, rovnako tak skontrolovať už existujúce názvy polí, pretože duplicity v tomto prípade nie sú povolené. Na obrázku 5 je zobrazené pridávanie nového vlastného poľa.

Obr. 5: Ukážka pridávania vlastného poľa

Ak užívateľ zvolí typ poľa zoznam, zobrazí sa dialógové okno, v ktorom musí užívateľ zadať možné hodnoty v pridávanom zozname. Každá z týchto hodnôt sa vloží na nový riadok textového poľa. Zadané hodnoty sú pri zmene textového poľa vyvolaním udalosti *OnChange* parsované využitím funkcie *split*, ktorá ako parameter prijíma reťazec „\n“. Týmto je zabezpečené načítanie údajov po jednotlivých riadkoch, ktoré sú následne uložené do *state* a pri odoslaní prostredníctvom tlačidla „uložiť“ zaslané na server. Zobrazené dialógové okno obsahuje taktiež jeden *checkbox* signalizujúci možnosť povoliť výber viacerých hodnôt v uloženom zozname. Po zaškrtnutí sa objaví hláška upozorňujúca užívateľa o tom, že po uložení tejto voľby ju nebude možné v budúcnosti vypnúť. Po potvrdení a validácii správnosti vložených hodnôt sú tieto údaje spolu s názvom a ostatnými atribútmi uložené. Ukážka vytvoreného okna na pridanie možných hodnôt zoznamu je na obrázku 6.



Obr. 6: Zobrazenie pridávania možných hodnôt vlastného poľa typu zoznam

V záverečnej časti úlohy trebalo pridať možnosť úpravy jednotlivých už vytvorených polí. K tomuto účelu som vytvoril novú triednu komponentu **FieldRowEdit**, ktorej úlohou bolo po kliknutí na odkaz „upraviť“ zobraziť riadok obsahujúci prvky *TextField* a *SelectField*, zobrazujúce aktuálne uložené údaje o konkrétnom užívateľskom poli s možnosťou následnej modifikácie povolených atribútov. Jediná položka, ktorú nie je možné zmeniť, je typ poľa. Zmena fronty prebieha prostredníctvom elementu *Select* umožňujúceho vybrať viacero možností. Samozrejme, pri všetkých úpravách prebieha rovnaká validácia údajov ako v prípade vytvárania nového poľa. Pri zázname s uvedeným typom „zoznam“ sa dajú možné hodnoty meniť po kliknutí ikony nachádzajúcej sa pri type vlastného poľa a k ich nastaveniam nie je potrebné kliknúť na tlačidlo „upraviť“. Po stlačení sa užívateľovi zobrazí rovnaké dialógové okno ako pri vytváraní nového poľa typu „zoznam“. Ak pri vytváraní alebo predchádzajúcej modifikácii nebol zaškrtnutý *checkbox* signalizujúci povolenie výberu viacerých hodnôt, je možné ho zaškrtnúť a túto zmenu uložiť.

V závere som ešte doladil vzhľad tohto nástroja pomocou CSS do výslednej podoby podľa požiadaviek spoločnosti. Táto úloha bola pre mňa v porovnaní s predchádzajúcimi náročnejšia najmä kvôli zložitej štruktúre dát prichádzajúcich do tejto komponenty, ktorú bolo potrebné pochopiť a tieto dáta správne spracovať. Náročné bolo tiež zamedziť všetkým možným chybám pri vytváraní a úprave polí, pretože scenárov nesprávneho vyplnenia užívateľom bolo veľa.

5 Zadanie hlavnej úlohy – Ganttov diagram

Po zvládnutí úvodných úloh, ktorých cieľom bolo zoznámenie sa s knižnicou React a celým systémom, som dostal zadanú hlavnú úlohu mojej praxe – vytvoriť nový interaktívny **Ganttov diagram**. V službe Projektově sa Ganttov diagram nachádzal, avšak nebol vytvorený prostredníctvom Reactu, jeho vykresľovanie nebolo dynamické a bolo príliš pomalé v dôsledku pribúdajúceho počtu užívateľov a vysokej záťaži servera. Bolo preto potrebné vytvoriť diagram, ktorý by všetky tieto výpočty presunul na klientské počítače, čím by sa výrazne urýchlilo jeho zobrazenie. Druhou výraznou nevýhodou starého diagramu bolo to, že nebol interaktívny. Dokázal zobraziť úlohy naplánované v čase, ale nebolo možné s úlohami priamo manipulovať prostredníctvom populárnej funkcie „chyť a pusť“ (angl. drag and drop).

5.1 Ganttov diagram

Gantov diagram je jedným z najpoužívanejších spôsobov zobrazenia aktivít v čase. Na horizontálnej ose sa nachádza zoznam aktivít, na vertikálnej ose sa navrchu nachádza hlavička s rovnako dlhými časovými úsekmi. Každá činnosť je zobrazená prostredníctvom jedného riadku a reprezentovaná obdĺžnikom vyjadrujúcim dátum začiatku úlohy, trvanie a dátum konca. To umožňuje užívateľovi tohto diagramu prehľadne vidieť kedy aktivity začínajú, kedy končia, ako dlho trvajú a kde sa prekrývajú. Tento nástroj sa používa najmä v projektovom riadení a poskytuje tiež pohľad na začiatok, priebeh a koniec projektu ako celku [9].

Prvý Ganttov diagram bol vytvorený v roku 1896 Karolom Adamieckim, ktorý ale tento nápad nespopularizoval vo svete. To urobil až o 15 rokov neskôr Henry Gantt, americký inžinier, ktorý predstavil svoju vlastnú verziu tohto diagramu, ktorá sa stala široko známa v západných krajinách. Spočiatku boli takéto diagramy pripravované ručne a pri každej zmene bolo potrebné prekreslenie. S rozvojom počítačov sa Ganttov diagram začal využívať prostredníctvom softvéru, kde môže byť jednoduchšie vytvorený, zmenený, zobrazený a ponúka viacero nástrojov pre efektívnejšie riadenie projektov [9].

5.2 Návrh riešenia

V tejto úlohe sa jednalo o zložitejšie a podstatne komplexnejšie zadanie, ktorého cieľom bolo vytvoriť jednu z hlavných častí celého systému. Preto bolo potrebné si riešenie vopred premyslieť. Po konzultácii so skúsenými vývojármi v spoločnosti bol navrhnutý akýsi základný postup, akým by mal vývoj tejto komponenty prebiehať.

Ako prvé bolo potrebné kompletne pripraviť „mriežku“ zobrazujúcu časovú os rozdelenú na primerané úseky, s možnosťou kompletného „zoomu“, teda zmeny pohľadu s rôznym stupňom priblíženia, resp. inými časovými úsekmi. Druhým krokom bolo spracovanie dát prichádzajúcich do tejto komponenty a ich následne zobrazenie – názvy konkrétnych úloh a obdĺžniky reprezentujúce trvanie zobrazených aktivít. To zahŕňalo aj zobrazenie rôznych typov úloh a vykreslenie

informácii o úlohách. Spôsob vykreslenia a detailnejší postup je opísaný v texte nižšie. Následne bolo potrebné zobraziť väzby medzi jednotlivými úlohami, ktoré reprezentujú závislosť jednej úlohy na úlohe inej. Až na záver riešenia sa pridala funkcionalita zabezpečujúca zmenu začiatkov a termínov úloh, zmenu ich trvania, vytvorenie väzieb ťahaním myšou a iné. Postupne bolo nevyhnutné jednotlivé časti vždy upraviť prostredníctvom kaskádových štýlov do požadovanej podoby, pričom hlavnou grafickou predlohou bol starý diagram s miernymi úpravami.

Predmetom návrhu tejto komponenty bol aj spôsob ukladania jednotlivých zmien na server. Nakoniec bol zvolený systém ukladania úprav do state tejto komponenty, rovnako tak vykresľovanie diagramu prebieha z vnútorných dát komponenty. Po vykonaní všetkých požadovaných zmien sú tlačidlom „uložiť“ odoslané na server.

5.3 Vytvorenie časovej mriežky s možnosťou zoomu

Na začiatku riešenia bolo potrebné nastaviť rozpätie dátumov, ktoré sa majú zobrazovať v záhlaví diagramu. To sa vykonáva v metóde *setBoundariesOfGantt*. Túto metódu som neskôr využil aj pri nastavovaní „hraníc“ diagramu pri zmenách zoomu, rovnako tak pri posune aktuálne zobrazovaných dátumov. Vzhľadom k záverečnému využitiu táto funkcia ako parametre prijíma počet mesiacov, ktoré majú byť zobrazené, úroveň posunu plátna *scroll*, ako aj premennú vyjadrujúcu to, či sa jedná o zavolanie metódy pri prvotnom zobrazovaní, alebo je metóda volaná ako reakcia na posun dátumového rozmedzia, resp. úrovne zoomu. Úlohou vyššie spomenutej metódy je nastaviť správne prvý a posledný deň aktuálne zobrazovaného diagramu podľa zvolenej miery priblíženia, pri zmene úrovne zobrazenia berie do úvahy aj aktuálny posun plátna obsahujúceho celý diagram. Zároveň pridáva každý zobrazený deň do poľa s názvom *dates*, ktoré neskôr slúži pri vykresľovaní hlavičky diagramu. Tieto dni sú do poľa pridané s využitím knižnice *Moment.js*.

Pri prvom vykreslení bolo podľa požiadaviek firmy potrebné v prípade, ak sa dátum aktuálneho dňa nachádza vo „vnútri“ projektu, nastaviť zobrazenie na čiaru dnešného dňa pomocou správneho posunu prvku *scroll*. V opačnom prípade sa tento prvok nastaví tak, aby bol viditeľný začiatok projektu. Pri zmene priblíženia sa plátno nastaví do správnej pozície podľa toho, kam bol diagram odsunutý v čase kliknutia na tlačidlo vyvolávajúce túto zmenu.

V momente, kedy sú už dáta ohľadom aktuálne zobrazovaných dátumov pripravené, je možné vykresliť hlavičku diagramu. Toto vykreslenie najprv prebiehalo v metóde *renderHeader*, neskôr bola z metódy vytvorená samostatná komponenta, ale spôsob vykreslenia zostal zachovaný. Spomenutá funkcia obsluhuje vykreslenie rôznych hlavičiek – zodpovedajúcich pre každý zoom. To je riešené prostredníctvom podmienok „if – else“, v ktorých sa podľa aktuálne zvolenej úrovne priblíženia vykonajú potrebné úpravy a výsledok funkcie sa vykreslí ako súhrnný *div* element obsahujúci rozdelenie na požadované časové úseky. V prípade mesačného zobrazenia sa kontroluje, či je vykreslený deň víkendom, a ak tomu tak je, táto hlavička je zafarbená šedou farbou pomocou podmienených štýlov. Šírka jednotlivých časových úsekov je vypočítaná s použitím vopred definovanej šírky jedného dňa. Najväčším problémom bolo vytvoriť hlavičku obsahujúcu názov mesiaca, pod ktorým je ešte tento mesiac rozdelený na jednotlivé týždne. V prípade, ak nejde

o prvý alebo posledný týždeň v mesiaci, sa jeho šírka vypočíta ako sedem dní vynásobených šírkou jedného dňa. V prípade prvého a posledného zobrazovaného týždňa ale bolo potrebné túto hodnotu vypočítať zložitejšie, čo je ukázané na výpise 8.

```
width: `${dates[dates.length - 1].day() === 0
      ? 7 * columnWidth
      : dates[dates.length - 1].day() * columnWidth}px`
```

Výpis 8: Ukážka výpočtu šírky hlavičky diagramu

Ďalším krokom je vykreslenie mriežky, do ktorej sa neskôr vykresľuje hlavný obsah diagramu. V čase prípravy tejto mriežky som pracoval so všetkými úlohami, ktoré mi do komponenty prišli ako *props* bez ich úpravy. K tomuto vykresleniu mi postačoval počet úloh, aby som vedel, koľko riadkov bude diagram obsahovať.

Vo funkcii s názvom *renderGrid* sa v cykle prechádzajú všetky úlohy vzťahujúce sa k zobrazenému projektu a ku každej sa podľa zvolenej úrovne priblíženia do poľa vloží potrebný počet elementov *div* zobrazujúcich jednu časovú jednotku. Celý princíp vloženia prebieha podobne ako pri vytváraní hlavičky diagramu. Najväčším rozdielom sú kaskádové štýly priradené jednotlivým elementom tak, aby sa vytvorili ohraničenia zobrazených riadkov, zvýraznenia víkendov a podobne. Ďalšou odlišnosťou je aj to, že sa v tomto prípade nevykresľuje a do elementov nevkladá žiaden text. Najnáročnejšou vecou na tejto metóde bolo upravenie jednotlivých elementov pomocou CSS tak, aby diagram vyzeral podľa požiadaviek, keďže na nastavenia jednotlivých rozmerov a umiestnenia elementov som využil rovnaké výpočty ako v metóde vykresľujúcej hlavičku.

Po zobrazení hlavičky a mriežky pre rôzne úrovne priblíženia nasledovalo pridanie možnosti kliknutím na príslušné tlačidlá zmeniť úroveň zoomu a zároveň umožniť posun zobrazovaných dátumov. Zmena priblíženia sa vykonáva v metóde *increaseZoom* (resp. *decreaseZoom*), ktoré sú volané ako reakcia na udalosť *onClick* na príslušnom tlačidle. V týchto metódach sa opäť prostredníctvom podmienok zisťuje, aká je aktuálne zvolená hodnota zoomu, podľa ktorej je nastavená šírka jedného stĺpca, začiatok a koniec zobrazovaného diagramu, ako aj úroveň posunu plátna. V prípade, ak už je priblíženie najvyššie (resp. najnižšie), metóda nevykoná žiadnu akciu. V opačnom prípade sa do *state* uloží hodnota zmeneného zobrazenia, rovnako tak vyššie spomenuté informácie.

Po vykonaní potrebných zmien je volaná metóda *setBoundariesOfGantt* s príslušnými parametrami, ktorá má za úlohu nastaviť nové zobrazované dátumy. Do tejto metódy sa v prípade zmeny úrovne priblíženia zasiela aj počet zobrazovaných mesiacov. To je pre jednotlivé úrovne vypočítané ako rozdiel dátumu konca poslednej úlohy v projekte a dátumu začiatku prvej úlohy. Ak by však počet zobrazovaných mesiacov dosiahol príliš malú hodnotu, je tento počet zvýšený na minimálny požadovaný, ktorý sa líši v závislosti od zvoleného priblíženia, viď výpis 9.

```
if (this.state.zoom === 'months') {
displayMonths = moment(this.state.tasks[0].dueDate).diff(this.state.tasks[0].
    startDate,'month') > 3
? moment(this.state.tasks[0].dueDate).diff(this.state.tasks[0].startDate,
      'month')
: 4
```

Výpis 9: Ukážka výpočtu počtu zobrazovaných mesiacov

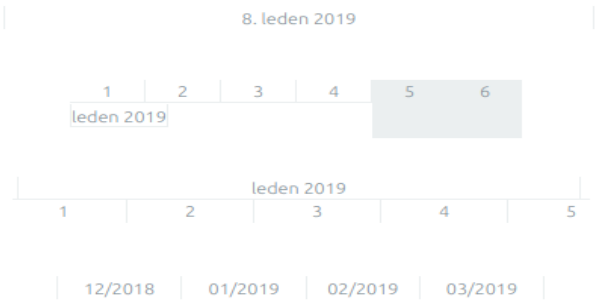
Metóda na nastavenie zobrazovaného diagramu prijíma pri tomto volaní aj úroveň posunutia plátna v čase stlačenia tlačidla tak, aby sa užívateľovi po priblížení, resp. oddialení zobrazila zodpovedajúca časť plátna a nebolo potrebné to znova posúvať. Táto hodnota je zistená pomocou funkcie *findDOMNode*, ktorá dokáže vrátiť DOM uzol s príslušnou referenciou aj s možnosťou zobrazenia dôležitých informácií o tomto elemente, vrátane rozmerov. Ja som použil v tomto prípade vlastnosť na zistenie úrovne posunutia s názvom *scrollLeft*, ktorý som vydelil aktuálnou šírkou jedného stĺpca v zobrazovanej mriežke, kvôli následným výpočtom pri nastavovaní úrovne plátna. Ukážka získania posunutia plátna je na výpise 10.

```
let scroll = findDOMNode(this.canvas).scrollLeft / this.state.columnWidth
```

Výpis 10: Ukážka zistenia úrovne posunu plátna

Dôvodom, že do funkcie *setBoundariesOfGantt* nezasielam iba číslo vyjadrujúce posun plátna tak, ako mi vráti vyššie spomenutá metóda je to, že ak by som po zmene úrovne priblíženia, a teda aj zmene rozmerov jednotlivých elementov nastavil tento posun na rovnakú hodnotu, nezobrazil by sa požadovaný časový úsek. Užívateľ by videl úplne iné dni ako pred priblížením, resp. oddialením. Takáto zmena úrovne priblíženia by nemala požadovanú funkčnosť a užívateľa by mohla zbytočne miasť. Preto sa táto úroveň naspäť prepočíta pri vykresľovaní nového zobrazenia až s už zmenenými rozmermi. Podobným spôsobom, ako zmena priblíženia, sa vykonáva aj posun zobrazovaných dátumov. V tomto prípade sa ale nemenia žiadne rozmery, iba dátum začiatku diagramu, ako aj počet zobrazovaných mesiacov, ktorý je rovnako ako pri zmene priblíženia, podľa potreby upravený na minimálnu požadovanú hodnotu.

Na záver tejto časti úlohy som do diagramu podľa aktuálneho dátumu vykreslil ešte čiaru dnešného dňa. Výsledné typy hlavičiek sú znázornené na obrázku 7.



Obr. 7: Ukážka rôznych typov hlavičiek diagramu

5.4 Spracovanie a zobrazenie úloh prichádzajúcich k zobrazovanému projektu

Keď už bola mriežka s dátumovou hlavičkou, zmenou priblíženia, ako aj zmenou dátumov hotová, ďalším krokom bolo upraviť prichádzajúce úlohy do požadovanej štruktúry a zobraziť ich v podobe obdĺžnikov vyjadrujúcich priebeh jednotlivých úloh v čase.

V objekte *props* mi do komponenty z úložiska prichádzajú všetky úlohy z databázy a nie iba tie, ktoré sa vzťahujú k zobrazovanému projektu, preto bolo potrebné najprv tieto úlohy nejakým spôsobom vyfiltrovať. To som urobil prostredníctvom už naprogramovanej funkcie *sortedVisibleTasksSelector*, do ktorej som zaslal kolekciu všetkých úloh prichádzajúcich z úložiska, kolekciu všetkých projektov a vopred preddefinovaný filter. Spomenutá metóda mi vráti úlohy patriace k zobrazovanému projektu zoradené podľa dátumu začiatku, ktoré potom zasielam do mnou vytvorenej funkcie *setTasksOrder*. V nej prebieha hlavná príprava dát pred samotným vykreslením a volaná je aj v prípade každej zmeny v dátach úložiska ovplyvňujúcej diagram do chvíle, kým neboli vykonané žiadne zmeny. To sa deje v jednej z metód životného cyklu Reactovej komponenty *componentWillReceiveProps* a jej zavolanie je podmienené zmenou potrebných dát prichádzajúcich z úložiska.

V metóde *setTasksOrder* je potrebné najprv úlohy zoskupiť tak, aby podúlohy nasledovali za svojimi rodičovskými úlohami a vytvoriť „stromovú“ štruktúru obsahujúcu hĺbku každej úlohy. To som dosiahol prostredníctvom už vytvorených metód *getSelfAndDescendants* a *buildProjectAndTasksTree*. To, čo mi táto metóda vráti, prechádzam prostredníctvom *map* a upravujem štruktúru každej úlohy. Keďže systém v súčasnosti neukladá na server začiatok a koniec projektu, ktorý som potreboval k jeho vykresleniu, v tejto metóde tiež nastavujem spomínané hodnoty. V prípade, ak je aktuálne pripravovaná úloha typu *„project“*, zistím najprv jeho podprojekty, keďže úlohy v nich obsiahnuté patria zároveň aj do hlavného projektu, ktorý tento podprojekt obsahuje. K stanoveniu začiatku každého projektu musím teda brať do úvahy aj tieto úlohy. Keď už mám zistené podprojekty, v prípade, ak ich aktuálne spracovávaný projekt obsahuje, vložím do poľa úlohy, ktorých id projektu je zhodné s nájdenými hodnotami. Potom do toho istého poľa pridám aj úlohy, ktoré majú id projektu zhodné s tým, ktorý je aktuálne spracovávaný. Keď už mám vytriedené úlohy, nájdem medzi nimi tie, ktoré začínajú najskôr, resp. končia najneskôr,

a ktoré určia začiatok a koniec celého projektu. Podobný výpočet vykonávam aj pri úlohách, ktoré nie sú projektom, ale iba v prípade, ak sú tzv. rodičovskou (angl. parent) úlohou, teda obsahujú podúlohy. Na záver pre každú úlohu upravím štruktúru tak, aby ako premenné obsahovala informácie potrebné k jej vykresleniu, kde zároveň vložím vyššie spomenuté vypočítané dátumy začiatku a konca projektov, resp. rodičovských úloh.

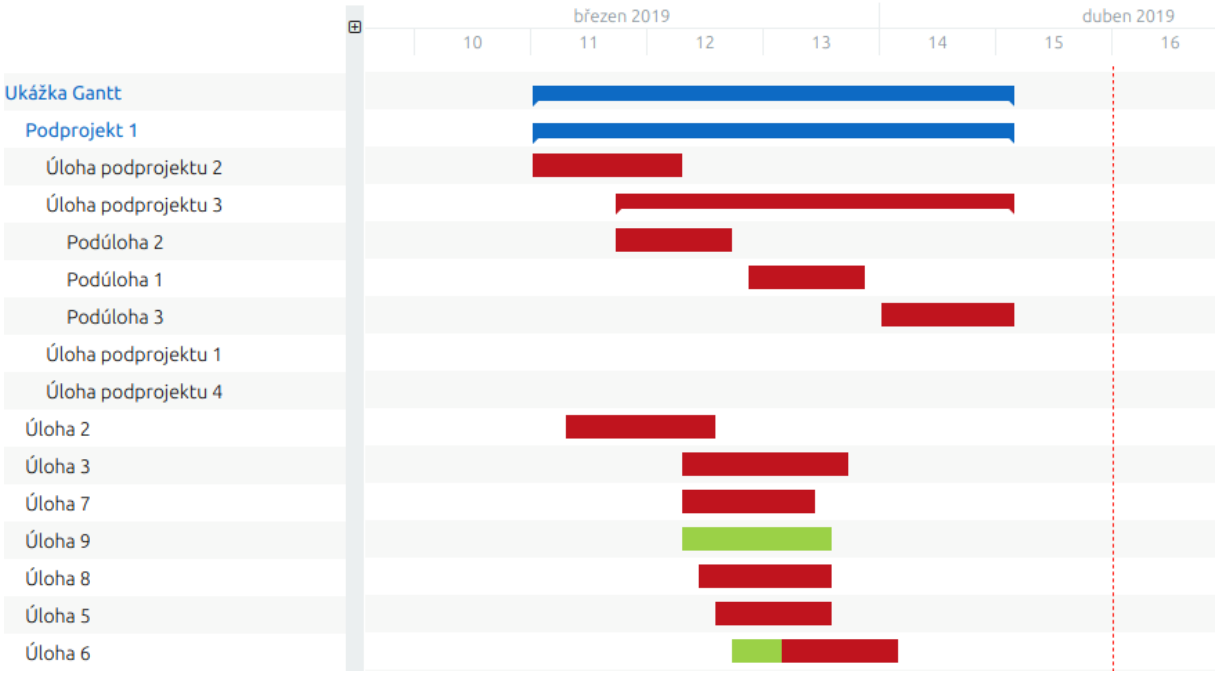
V závere metódy si do *state* uložím spracované úlohy v upravenej štruktúre, z ktorej sa potom vykresľujú obdĺžniky v diagrame, rovnako tak si ich uložím v štruktúre vrátenej z funkcie *buildProjectAndTasksTree*, z ktorej hĺbku jednotlivých úloh využívam pri vykreslení názvov. Týmto sú dáta plne pripravené na vykreslenie diagramu.

Ako prvé som vytvoril vykreslenie názvov úloh, pretože si to nevyžadovalo až toľko práce ako neskoršie vykresľovanie obdĺžnikov. V metóde *renderTasks* prechádzam stromovú štruktúru úloh s názvom *flattenTreeWithDepth*, ktorú mám uloženú v state a do poľa *taskRows* pridávam jednotlivé názvy úloh v štruktúre tabuľky. Podľa príslušnej hĺbky úlohy nastavujem aj *padding* názvu tak, aby bola zohľadnená stromová štruktúra. V prípade projektu je názov zobrazený modrou farbou. Po kliknutí na názov úlohy sa jej detail zobrazí v dialógovom okne, ak ide o podprojekt, zobrazí sa celý diagram s týmto podprojektom v „úlohe“ hlavného projektu.

Vykresľovanie obdĺžnikov sa vykonáva v samostatnej komponente s názvom **Bar**, ktorej inštancie sú vytvárané v hlavnej komponente, a ktorá prijíma potrebné dáta prostredníctvom *props*. K tomuto vytvoreniu slúži metóda *renderBars*, v ktorej sa pomocou cyklu prechádzajú všetky úlohy uložené v state *tasks* v požadovanej upravenej štruktúre, a ku každej úlohe je vytvorená príslušná komponenta **Bar**. Tá je následne vložená do poľa. Celé vykreslenie, úprava dát a potrebné výpočty prebiehajú práve v tejto komponente, ktorej výsledkom je vykreslenie príslušných obdĺžnikov, ako aj celá rada ďalšej funkcionality. K vykresleniu obdĺžnikov som využil formát SVG, ktorý v sebe uchováva tiež kruhy využívané pri vytváraní väzieb medzi jednotlivými úlohami. Formát SVG bol zvolený hlavne kvôli potrebe vykreslenia rôznych tvarov (kruh, čiara), ktoré je prostredníctvom elementov *div* zložité vytvoriť. Taktiež v budúcnosti v prípade úpravy na HTML canvas bude táto zmena jednoduchšia. Spomenutý formát je navyše použitý aj na iných miestach v systéme k vykresleniu napríklad myšlienkovej mapy, takže sa tým zachovala jednotnosť. Element obsahujúci obdĺžnik som musel umiestniť tak, aby sa nachádzal presne v mieste trvania konkrétnej úlohy. To som dosiahol vypočítaním hodnoty umiestnenia *top*, pri ktorom som vychádzal z vopred definovanej výšky jedného riadku a čísla aktuálne vykresľovanej úlohy. K získanému číslu som pripočítal ešte výšku hlavičky, keďže samotné vykresľovanie celého diagramu prebieha do rovnakého elementu ako vykreslenie hlavičky. K zisteniu horizontálneho umiestnenia som využil dáta prichádzajúce z nadradenej komponenty vo forme *props*, konkrétne hodnotu šírky stĺpca *columnWidth*, vykresľované dátumy *dates* a dátum začiatku vykresľovanej úlohy. Na záver bolo k tomuto elementu potrebné vypočítať aj jeho šírku, keďže kvôli neskoršiemu vykresľovaniu väzieb nebolo možné tento element nechať vykreslený až do konca riadku. Pri tomto výpočte som musel do úvahy brať aj situáciu, ak úloha síce začína v deň, ktorý je aktuálne zobrazovaný na diagrame, ale jeho termín aktuálne zobrazený nie je. Takáto úloha sa v diagrame

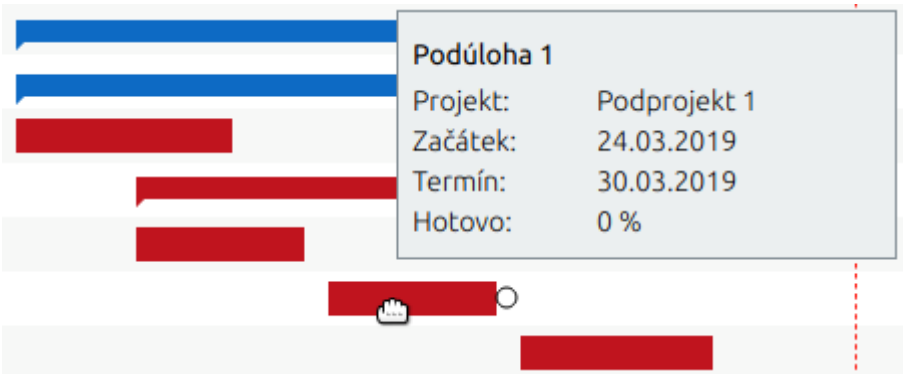
musí zobraziť, avšak nemôže presiahnuť plochu diagramu a byť vykreslená mimo vytvorenej mriežky. Túto nepríjemnosť som vyriešil overením, či rozdiel dátumu začiatku úlohy a prvého zobrazovaného dátumu v diagrame vynásobený rozmerom jedného dňa s pripočítaným trvaním úlohy je menší ako celkový rozmer vykreslenej mriežky. Ak áno, vykreslí sa príslušná úloha v celom jej trvaní, v opačnom prípade sa vykreslí iba do posledného zobrazovaného dňa.

Do vyššie opísaného SVG elementu som vložil element *rect*, ktorý umožňuje vytvorenie obdĺžnika definovaním jeho umiestnenia a rozmerov. Nakoľko som výpočtami zistil už správne umiestnenie príslušného SVG elementu, tu bolo možné jednotlivé rozmery pevne definovať podľa požiadaviek. Vzhľadom k tomu, že sa jednotlivé úlohy svojim vykresleným tvarom líšia podľa toho, či sa jedná o obyčajnú úlohu, tzv. „obálkovú úlohu“ alebo projekt, vykresľovaný obdĺžnik sa líši hlavne parametrami *„y“*, *„height“* a *„fill“*, ktorý reprezentuje farbu vyplnenia elementu. Pri vykresľovaní bolo v prípade úloh potrebné tiež zohľadniť a farebne odlíšiť stav vykonávania úlohy podľa premennej „% hotovo“. K tomuto účelu som pridal elementy typu *rect* dva. Na spodok som umiestnil pre každú úlohu obdĺžnik červenej farby vyjadrujúci nesplnenú časť úlohy, nad ňu sa následne zelenou farbou vykreslí obdĺžnik, ktorého dĺžka je vypočítaná v závislosti na vyššie spomenutej premennej. V prípade projektu alebo „obálkovej úlohy“ sa k tomuto obdĺžniku pridá na okraje ešte malý polygón vizuálne vyjadrujúci skutočnosť, že sa jedná o úlohu (resp. projekt), ktorá v sebe obsahuje úlohy iné. V prípade polygónu sa ako parameter *points* zadávajú body, ktorých spojením vznikne požadovaný tvar. K ich výpočtu som využil dĺžku a výšku obdĺžnika zobrazujúceho úlohu. Vykreslené úlohy je možné vidieť na obrázku 8.



Obr. 8: Ukážka vykreslených obdĺžnikov reprezentujúcich jednotlivé úlohy

Po zobrazení úloh bolo potrebné pridať ešte nejakú „tabuľku“ s informáciou o úlohe. To som vyriešil pridaním nového *div* elementu už mimo SVG, do ktorého som zobrazil názov úlohy, dátum začiatku, termín a percento už vykonanej časti úlohy. Tento element sa vykreslí iba v prípade vstupu kurzorom myši do priestoru konkrétneho obdĺžnika úlohy vo vypočítanej pozícii. Pri výpočte umiestnenia informácií som použil pôvodné umiestnenie SVG elementu, ktoré som posunul o pevnú čiastku nadol, resp. nahor podľa toho, či sa úloha nachádzala vo vrchnej alebo spodnej časti diagramu. Horizontálne umiestnenie informácií o úlohe som vypočítal na základe aktuálnej pozície kurzora tak, aby bola oproti vstupu do obdĺžnika posunutá o pevnú hodnotu. Aj tu prebieha kontrola toho, či sa kurzor nenachádza na okraji plátna, a ak by ho zobrazované informácie mali presiahnuť, zobrazia sa na opačnej strane od kurzora. Pozíciu kurzora zisťujem na príslušnom SVG elemente pomocou udalosti *onMouseEnter*, kde zároveň nastavím state *visibleInfo* na hodnotu *true*. Na obrázku 9 sa nachádza výsledné zobrazenie detailu úlohy.



Obr. 9: Zobrazenie detailov o úlohe

Vzhľadom k tomu, že sa v systéme pri projektoch ukladá plánovaný začiatok a plánovaný termín, v prípade vykresľovania projektu som pridal ešte druhý SVG element. Do neho sa vykreslí tenký obdĺžnik šedej farby nad zobrazovaným projektom vtedy, ak má tento projekt vyplnené oba tieto dátumy.

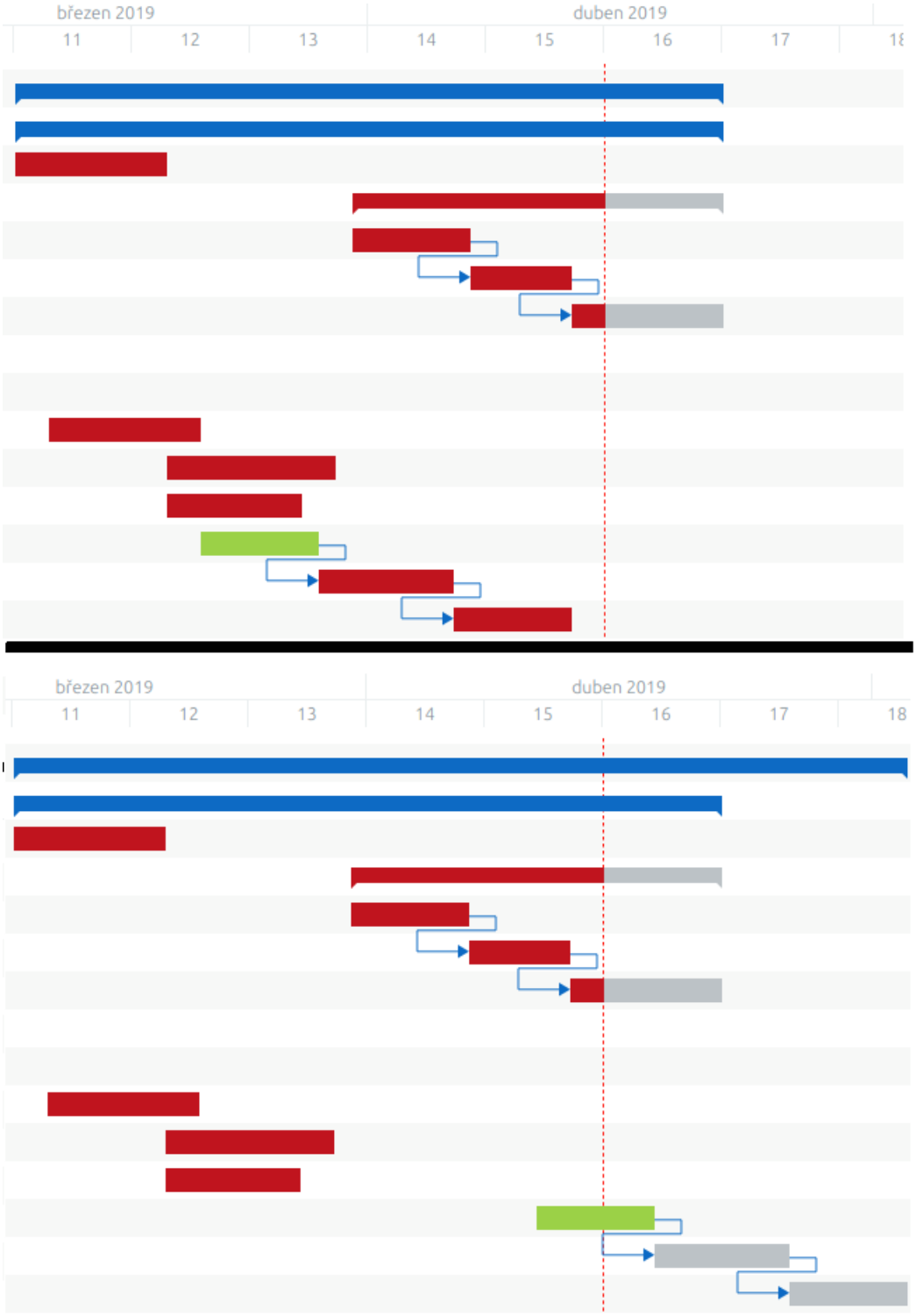
V nasledujúcich riadkoch opíšem vytváranie funkcionality posúvania a „naťahovania“ úloh napriek tomu, že som ich vytváral až ako jednu z posledných častí, nakoľko sa tieto časti kódu nachádzajú v rovnakej komponente **Bar**. K tomu, aby som mohol jednotlivé úlohy posúvať a tým meniť ich umiestnenie v čase, musel som do SVG elementu obsahujúceho príslušný obdĺžnik pridať atribút *draggable*, aby bolo možné ním pohybovať. Na elementoch *rect* vykresľujúcich úlohy som využil udalosť *onMouseDown*, ktorá po stlačení tlačidla myši na tento obdĺžnik nastaví do state pozíciu kurzora pred začatím „ťahania“, rovnako tak príznak *isDragging*. To sa vykoná iba v prípade, ak je aktuálne prihlásený užívateľ manažérom vo všetkých zobrazených projektoch. V opačnom prípade je celý diagram v režime „pre čítanie“ a nie je možné v ňom vykonávať žiadne zmeny. Akonáhle sa state *isDragging* nastaví na hodnotu *true*, dochádza k pridaniu nového SVG elementu, ktorý slúži výlučne pre potreby posunutia tejto úlohy a je „natiahnutý“ na rozmer celého plátna obsahujúceho diagram. Na pridanom elemente využívam udalosť *on-*

MouseMove, pri ktorej dochádza k prepočtu umiestnenia úlohy a nastaveniu nového dátumu začiatku a termínu. Podľa aktuálnej pozície kurzora a pozície pred začatím „ťahania“ uloženej v *state* sa zistí, ktorým smerom je úloha posúvaná a podľa toho dochádza k výpočtu nových dátumov. V prípade, ak je aktuálna pozícia kurzora mimo plátna, zastaví sa presúvanie úlohy. V závere sa aktuálna pozícia kurzora uloží do state ako pozícia pred ťahaním, ktorá je zasa v ďalšom volaní využitá k výpočtom. Rovnako tak využívam udalosť *onMouseUp*, ktorá zabezpečí pri ukončení ťahania nastavenie nových dátumov v rodičovskej komponente v state *tasks*. Tu je volaná metóda rodičovskej komponenty *setNewTasksDays* zaslaná v *props*, kam sa odošle číslo úlohy, nový začiatok a nový termín.

Vo vyššie spomenutej najzložitejšej metóde celého diagramu dochádza k nastaveniu dátumov nie len posúvanej úlohy, ale aj úloh iných, ktoré sú týmto posunom ovplyvnené. V nej sa posudzujú aj neskôr opísané väzby medzi úlohami, ktoré signalizujú závislosť úloh nimi spojených. Postupne sa teda najprv rekurzívnou metódou prechádzajú väzby do posúvanej úlohy smerujúce, následne väzby vychádzajúce z tejto úlohy. K posunu úloh prepojených väzbami dôjde iba v prípade, ak rozdiel nového začiatku (resp. termínu) posúvanej úlohy a starého začiatku (resp. termínu) úlohy napojenej väzbou, je menší ako jeden deň. V praxi to napríklad znamená, že ak posúvame úlohu, z ktorej vychádza väzba do inej úlohy, s termínom päť dní pred začiatkom naväzujúcej úlohy, o dva dni, k posunu ovplyvnenom spojením väzbou v tomto prípade nedôjde. Ak ale rovnakú úlohu posunieme napríklad o šesť dní, dôjde k posunu úlohy napojenej väzbou tak, aby jej začiatok bol presne nasledujúci deň po užívateľom posúvanej úlohe, teda o dva dni. V tejto metóde sa nastavujú aj prípadné obálkové úlohy obsahujúce úlohu posunutú v dôsledku napojenia väzbou. V prípade, ak je úloha posunutá v dôsledku napojenia väzbou obálková, dochádza k nastaveniu úloh, ktoré obsahuje. Tieto metódy sú volané aj z vyššie spomenutej funkcie *setNewTasksDates*, kde ale dochádza k nastavovaniu „rodičov a potomkov“ úlohy posúvanej užívateľom. Na záver prebieha ešte nastavovanie nových dátumov projektu. Opísaná metóda je tiež jedným z miest, kde sa nastaví inštančná premenná *modified* na true. Od tejto chvíle sa vykreslenie diagramu riadi výlučne dátami zo *state* a nie dátami prichádzajúcimi z *props*. Takáto zmena sa vykonáva aj pri úprave trvania úlohy a vytvorení či odstránení väzby.

Ďalšou možnosťou zmeny úloh je zmena ich trvania. Kvôli podpore tejto funkcie som na vykreslené obdĺžniky dodal na ich pravý a ľavý okraj ďalší malý element typu *rect*, nad ktorým sa kurzor zmení na šípku vyjadrujúcu možnosť „naťahovať“ príslušnú úlohu do konkrétnej strany (pravá šípka v prípade naťahovania úlohy k dátumom v budúcnosti, ľavá v prípade dátumov v minulosti). Po kliknutí na toto miesto sa podľa príslušnej strany kliknutia zavolá metóda, ktorá opäť nastaví pozíciu kurzora pred zmenou, rovnako tak uloží informáciu o tom, či je úloha „naťahovaná“ do pravej alebo ľavej strany. Následne sa opäť zobrazí rovnaký SVG element ako v prípade presúvania celej úlohy. Podľa zvolenej strany naťahovania sa mení iba jeden z dátumov – začiatok alebo termín, čím dochádza k zmene trvania úlohy. Pri ukončení vykonávanej zmeny je znova volaná rovnaká metóda ako v prípade presúvania úlohy, ktorá tým istým spôsobom nastaví dátumy všetkým ovplyvneným úlohám. V závere bolo potrebné ešte upraviť zobrazované

informácie v priebehu posunu alebo naťahovania úlohy, kde sa už nezobrazuje informácia o tom, koľko percent úlohy je už splnených. Pri naťahovaní sa navyše zobrazuje nové trvanie úlohy, aby mal užívateľ lepší prehľad o tom, ako túto úlohu mení. Ukážka posúvania je na obrázku 10.



Obr. 10: Úlohy zobrazené pred (hore) a po posune (dole)

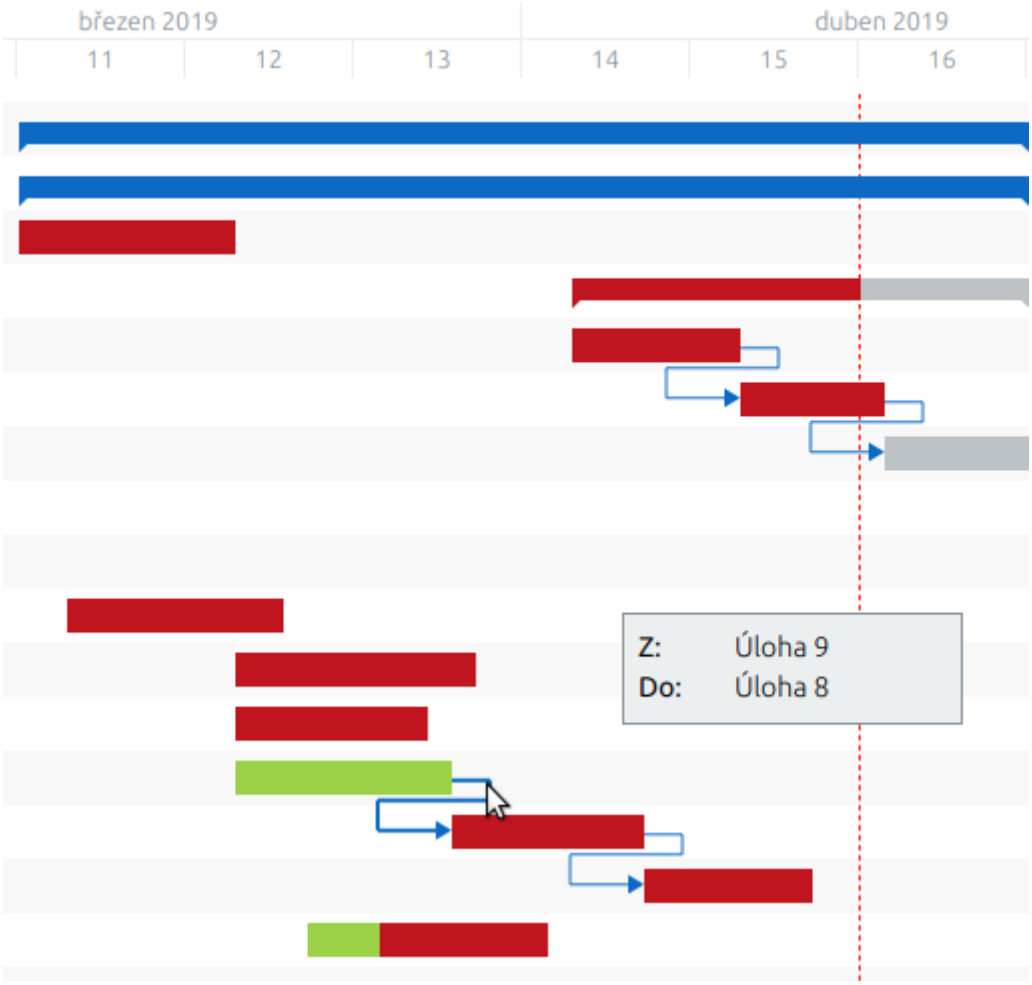
5.5 Vykreslenie väzieb s vytvorením príslušnej funkcionality

Ďalším krokom bolo vykreslenie väzby medzi jednotlivými úlohami. Tie v komponente sťahujem priamo zo servera a ukladám do state s názvom *relations* bez akejkoľvek úpravy. Samotné vykreslenie je zabezpečené samostatnou komponentou s názvom **Relation**, ktorú vytváram prechádzaním všetkých väzieb a odoslaním potrebných dát. Pred vykreslením každej väzby ešte overujem, či sa jedná o väzbu medzi vykreslenými úlohami. Môže totiž nastať situácia, že úlohy, ktoré nemajú vyplnený začiatok a termín súčasne, síce svoj riadok v diagrame majú, no pri týchto úlohách nie je vykreslený obdĺžnik. V uložených dátach sa podobné väzby môžu nachádzať. V takomto prípade by mohla väzba začínať alebo končiť v prázdnom riadku mimo úlohy. Ak je táto podmienka splnená, do poľa je vložená príslušná inštancia komponenty **Relation**. Všetky väzby sú potom umiestnené na novom SVG elemente nachádzajúcom sa na celej ploche diagramu.

Samotná komponenta **Relation** je zodpovedná za vykreslenie príslušnej väzby, zobrazenie informácií o väzbe po vstupe kurzora do jej priestoru, ako aj odstránenie väzby dvojitým klikom myšou. Jednou z najnáročnejších vecí bolo vypočítať správne body väzieb tak, aby začínali presne za obdĺžnikom úlohy a šípkou končili priamo pred úlohou, kam väzba smeruje. Priamo pri vykreslení bolo potrebné ešte zistiť, ktorým smerom bude väzba smerovať. Jednotlivé čiary som vykreslil prostredníctvom elementu *path*, ktorý v atribúte *d* prijíma reťazec obsahujúci dva body tvoriace vykresľovanú čiaru. V prípade, ak ide o väzbu medzi úlohami, kde úloha pri začiatku väzby končí skôr a medzi jej koncom, a začiatkom úlohy na konci väzby je nejaký posun, vykreslia sa takéto čiary tri. V opačnom prípade sa čiar vykreslí päť. To je spôsobené tým, že ak je začiatok nasledujúcej úlohy presne nasledujúci deň po termíne tej predchádzajúcej, alebo dokonca ešte skôr, dosiahnuť toto zobrazenie troma čiarami by nebolo možné. Veľkosť posunu, podľa ktorého sa rozhodne, o akú väzbu ide, sa líši v závislosti od zvolenej úrovne priblíženia. Tento posun bol zvolený testovaním tak, aby tieto väzby vyzerali čo najlepšie. K výpočtom bodov príslušnej čiary využívam dátumy úloh, rozmery jedného dňa a pevne definované posuny. Na konci väzby vykreslím ešte polygón, ktorý jej dodá požadovaný vzhľad šípky s určeným smerom.

Po vykreslení väzieb som potreboval vyriešiť zobrazenie informácií o väzbe po vstupe kurzora na túto väzbu, rovnako tak jej odstránenie a zvýraznenie vykreslením čiar s vyššou hrúbkou. Vzhľadom k tomu, že sú tieto čiary reprezentujúce väzby príliš tenké, reakcia na vstup do tejto oblasti nebola použiteľná. Preto som sa rozhodol, že ku každej čiare tvoriacej väzbu pridám ešte neviditeľný obdĺžnik, v ktorom zareagujem na vstup kurzora, čím zlepším ovládanie potrebných funkcií. K tomu som využil element *<g>*, ktorý funguje ako akýsi kontajner pre uchovávanie viacerých SVG elementov. V týchto novo pridaných obdĺžnikoch reagujem na udalosť *onMouseEnter*, v ktorej zobrazím informácie o konkrétnej väzbe podobne, ako v prípade úloh. Rovnako tak zvýšim atribút *strokeWidth* uložený v state, ktorý charakterizuje hrúbku vykreslených čiar. V udalosti *onMouseLeave* zobrazenie informácií o väzbe ukončím a hrúbku čiar vrátim na pô-

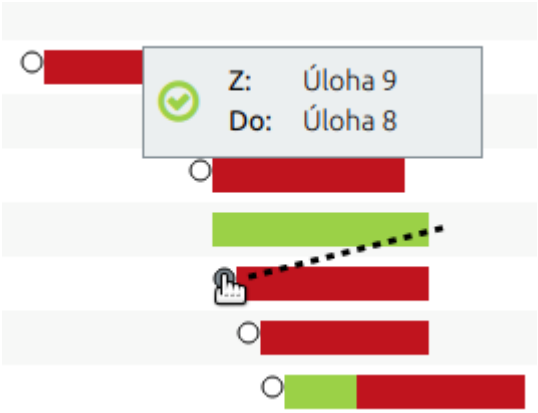
vodnú hodnotu. Posledná udalosť, ktorú tu využívam, je udalosť *onDoubleClick*. Tá v prípade, že je užívateľ manažérom vo všetkých zobrazených projektoch, zobrazí potvrdzovacie dialógové okno. Ak užívateľ potvrdí túto voľbu je príslušná väzba odstránená zo state hlavnej komponenty a pridaná do poľa *deletedRelations*, čo súvisí s neskorším ukladaním zmien na server. Na obrázku 11 je možné vidieť zobrazené väzby.



Obr. 11: Ukážka vykreslenia väzieb so zobrazeným detailom o konkrétnej väzbe

Po zobrazení väzieb s možnosťou ich odstránenia chýbalo pridanie funkcie tieto väzby vytvárať, keďže to nie je možné urobiť na žiadnom inom mieste v systéme. K tomu slúžia malé kruhy, ktoré sa zobrazia na okrajoch obdĺžnika príslušnej úlohy po vstupe do jej priestoru myšou. Tie som pridal do vyššie opísanej komponenty **Bar**. K tomu som využil element *circle*, ktorému stačí zadať bod stredu a požadovaný polomer. Prostredníctvom udalostí po vstupe kurzora do oblasti tohto kruhu zmením jeho farbu na šedú, aby užívateľ lepšie postrehol možnosť nejakej funkcionality. Po kliknutí na kruh sa zavolá metóda rodičovskej komponenty, ktorá do jej *state* uloží číslo úlohy, odkiaľ je väzba vytváraná. Tam sa uloží aj pozícia kurzora na plátne diagramu, ktorá je potrebná k vykresľovaniu šípky indikujúcej užívateľovi kam túto väzbu vytvára. V me-

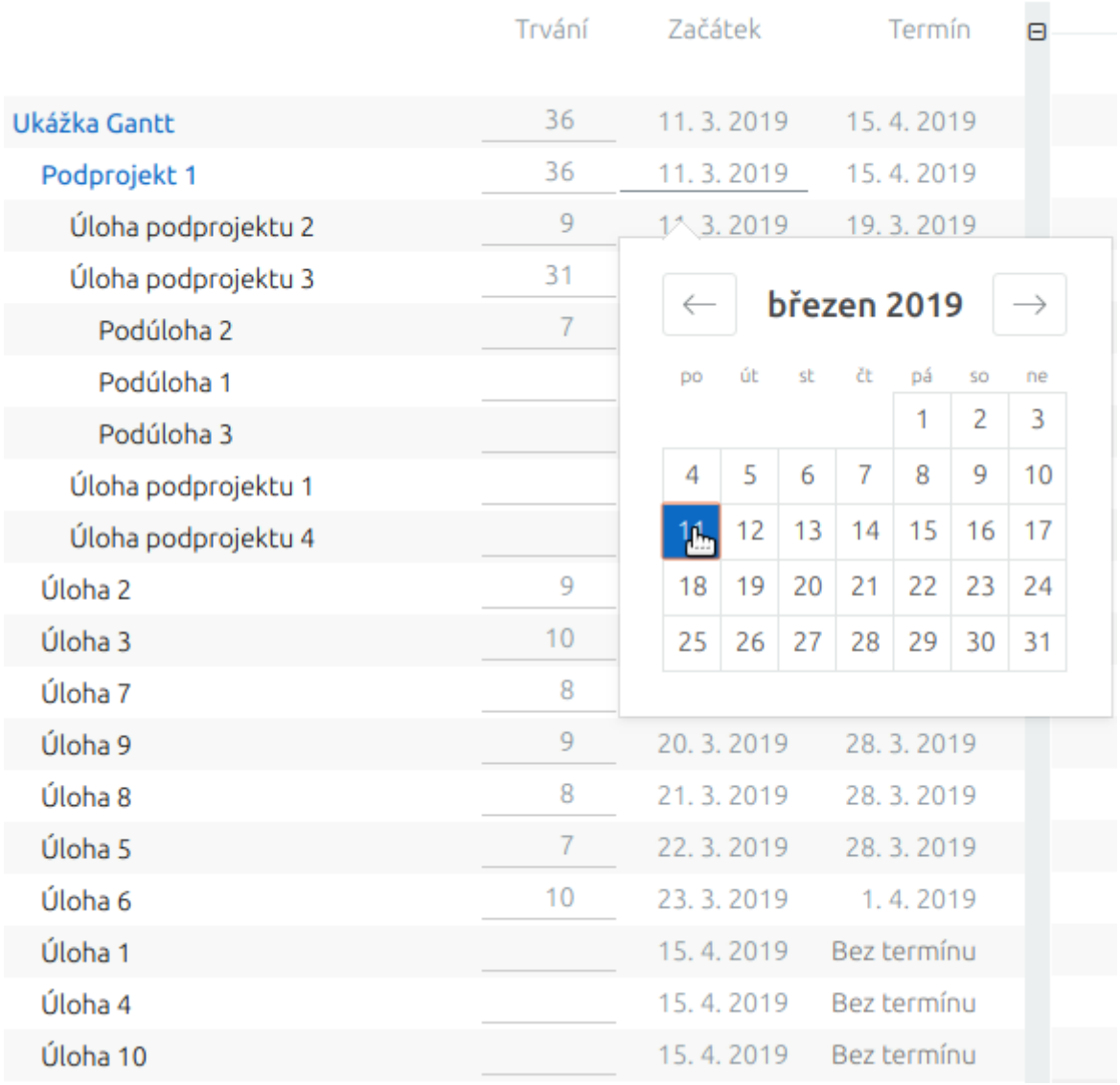
tóde *drawRelation* sú vytvárané inštancie samostatnej komponenty **Circle**. Túto komponentu som vytvoril k vykresleniu kruhov, v ktorých je možné prebiehajúce vytváranie väzby ukončiť, a tým vytvoriť väzbu do úlohy nachádzajúcej sa pri tomto kruhu. Zároveň v tejto metóde dochádza k vykresleniu šípky. Prvý bod šípky je vytvorený prostredníctvom pozície nastavenej pri začatí vytvárania väzby. Druhý bod sa určí podľa aktuálnej pozície kurzora. Vytváraná väzba je vykreslená v samostatnom SVG elemente, nad ktorým sú vykreslené ešte vyššie spomenuté inštancie komponenty **Circle**. Táto komponenta sa stará o zobrazenie kruhov iba pri tých úlohách, kam je možné väzbu vytvoriť. Ak napríklad užívateľ vytvára väzbu z obálkovej úlohy, kruhy na dokončenie tejto akcie sa pri jej podúlohách nezobrazia, nakoľko takáto väzba nie je možná. Udalosť *onMouseUp* v tejto komponente zabezpečí výpočet nových dátumov pre úlohu, kam bude nová väzba smerovať, keďže požiadavkou bolo pri vytvorení väzby nastaviť dátum začiatku na nasledujúci deň po termíne úlohy, odkiaľ väzba smeruje. Na záver je táto väzba pridaná k ostatným väzbám v hlavnej komponente a dochádza opäť k posunu dátumov nie len úlohy na konci väzby, ale aj všetkých úloh, ktoré tento posun ovplyvňuje. Na obrázku 12 je ukázané vytváranie väzby.



Obr. 12: Vytváranie väzby pomocou myše

5.6 Záverečné úpravy

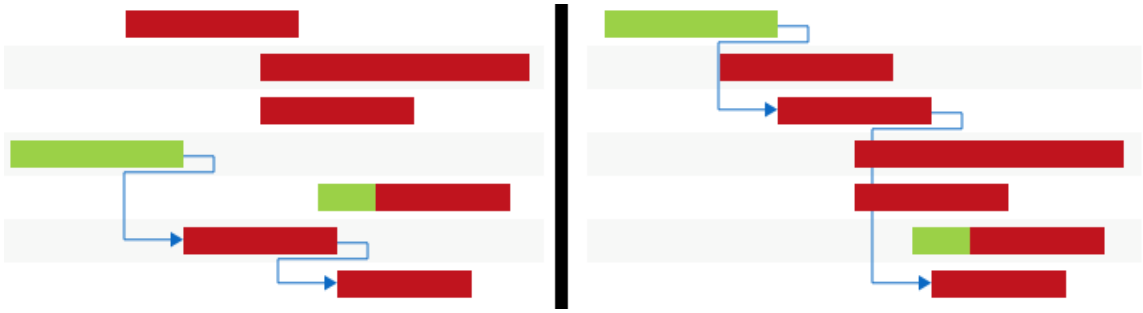
Po úspešnom vytvorení takmer kompletnej funkcionality som medzi názvy úlohy a samotný diagram pridal tenkú lištu smerujúcu od hora dole, ktorá po kliknutí zobrazí nové stĺpce, ktoré umožňujú úpravu začiatku, termínu a trvania úlohy iným spôsobom. Pri tomto nastavení môže užívateľ jednoducho zmeniť dátum po kliknutí naň, kedy sa užívateľovi zobrazí malý kalendár, z ktorého si deň jednoducho zvolí. Pri takejto forme úpravy dátumov sa vykonávajú rovnaké zmeny všetkých ovplyvnených úloh tak, ako v prípade zmien funkciou drag and drop a využívajú sa pri tom tie isté funkcie. Ukážka tohto vylepšenia sa nachádza na obrázku 13.



Obr. 13: Úprava dátumov výberom z kalendára

Keďže niektoré veľké firmy majú projekty obsahujúce veľké množstvo úloh, pridal som možnosť skryť podúlohy. Takáto voľba užívateľa sa pri prvom spustení nastaví na vypnutú, na zapnutú sa ale nastaví v prípade, ak sa v otvorenom diagrame nachádza viac ako päť podprojektov. Pri ďalšom otvorení diagramu sa pre toto nastavenie použije naposledy zvolená hodnota, ktorá sa pri každej zmene uloží do local storage.

Po vykonaných úpravách úloh – ich posune a zmene začiatkov resp. termínov som do diagramu pridal tlačidlo, po stlačení ktorého dôjde k zoradeniu úloh podľa dátumu začiatku, čím sa obnoví požadovaný vodopádový model zahŕňajúci tieto zmeny, viď obrázok 14.



Obr. 14: Úlohy pred zoradením (vľavo) a po zoradení (vpravo)

V úplnom závere bolo potrebné zaistiť ukladanie zmien na server, bez ktorého by boli všetky vykonané úpravy zbytočné a pri znovuotvorení diagramu, prípadne konkrétnych úloh na inom mieste systému, by sa neprejavili. Na to slúži tlačidlo „uložiť“, ktoré po stlačení vykoná odoslanie všetkých zmien na server, a až vtedy sú o zmenách informovaní členovia projektu. V metóde *onSave* sa najprv pripraví požadovaná štruktúra úloh a väzieb, ktoré sa následne odošlú na server metódou *HTTP PUT*.

5.7 Zhodnotenie

Táto úloha ma veľa naučila a v súčasnosti je nový Ganttov diagram testovaný vybranými užívateľmi systému pred tým, ako bude plošne nasadený pre všetkých. Oproti starému diagramu došlo k výraznému zrýchleniu a zlepšeniu používania, čo potvrdili aj testujúci užívatelia. Veľkým prínosom je jednoduchá úprava všetkých úloh na jednom mieste. Vytvoriť tento diagram trvalo značný čas a vyžadovalo si napriek základným skúsenostiam s vývojom v Reacte značné doštudovanie technologií, s ktorými som sa v predchádzajúcich úlohách nestretol. V budúcnosti bude možné pridať ďalšie funkcie, ako napríklad pridanie novej úlohy, filter úloh a čokoľvek, čo sa ukáže byť pre užívateľov prospešné.

6 Zhodnotenie uplatnených a chýbajúcich znalostí

6.1 Teoretické a praktické znalosti získané v priebehu štúdia uplatnené v priebehu praxe

Pri vykonávaní praxe som využil znalosti predovšetkým z programovacích predmetov, akými boli Algoritmy I, Algoritmy II, Programovanie I, Programovanie II, Vývoj informačných systémov, Programovací jazyky I, Programovací jazyky II, ktoré ma naučili hlavne spôsob myslenia v programovaní. Nezáleží na tom, v akom jazyku človek programuje, pretože základné princípy sú všade rovnaké. Takisto som využil znalosti z predmetu Tvorba aplikací pro mobilní zařízení I, kde som získal základy HTML. Dobre mi poslúžili aj znalosti z predmetu Uživatelská rozhraní, vďaka čomu som dokázal lepšie vytvoriť vhodné rozhranie pri mne zadaných úlohách, aj keď do úvahy bolo potrebné vziať aj vzhľad aplikácie ako celku, rovnako tak požiadavky zo strany firmy.

6.2 Znalosti chýbajúce v priebehu praxe

To, čo mi najviac chýbalo pri vykonávaní mojej praxe bola znalosť knižnice React, prípadne aspoň lepšia znalosť jazyka JavaScript. Takisto mi chýbala znalosť kaskádových štýlov a skúsenosť s verzovacím systémom Git, ktorý sa využíva vo väčšine spoločností zaoberajúcich sa vývojom softvéru.

7 Celkové zhodnotenie praxe

V závere musím zhodnotiť, že voľba absolvovať bakalársku prácu formou praxe bolo skvelé rozhodnutie. Vo firme Projektově ma naučili veľa užitočných vecí, ktoré určite využijem v mojom ďalšom profesnom živote. Mal som možnosť podieľať sa na vývoji reálneho produktu s využitím najnovších technologií, ktoré sa dnes v praxi využívajú. Po každej ukončenej úlohe mi bolo vysvetlené, čo by sa dalo v budúcnosti urobiť lepšie. V tejto firme sa snažili o to, aby som sa niečo naučil a nie iba splnil povinnú úlohu. Rovnako veľkým prínosom ako zisk skúseností a nových vedomostí je to, že mnou vytvorené komponenty uľahčia a spríjemnia užívateľom používanie systému.

Väčšina úloh opísaných v tejto práci je v súčasnosti využívaná užívateľmi služby Projektově, ostatné sú testované. Po dôkladnom odskúšaní užívateľmi budú nasadené pre všetkých, prípadne je možný vývoj ďalších rozšírení, hlavne v prípade Ganttovho diagramu.

Literatúra

[1] Interná dokumentácia firmy.

[2] ReactJS [online]. Dostupné z: https://reactjs.org/

[3] Virtual DOM [online]. Dostupné z: https://hackernoon.com/virtual-dom-in-reactjs-43a3fdb1d130

[4] Flux [online]. Dostupné z: https://facebook.github.io/flux/

[5] Immutable.js zdroj 1 [online]. Dostupné z: http://untangled.io/immutable-js-an-introduction-with-examples-written-for-humans/

[6] Immutable.js zdroj 2 [online]. Dostupné z: https://immutable-js.github.io/immutable-js/

[7] Lodash [online]. Dostupné z: https://lodash.com/

[8] Moment.js [online]. Dostupné z: https://neoteric.eu/blog/why-should-you-use-moment-js-and-how-to-do-it-effectively/

[9] Gantt Chart [online]. Dostupné z: https://www.gantt.com/